VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Simulátor robotů s využitím frameworku PhysX

PhysX Based Robot Simulation

2010

Rostislav Žídek

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2010 .....

Rád bych na tomto místě poděkoval především svému vedoucímu za odborné vedení a tipy při zpracování této diplomové práce. Další poděkování pak směřují ke společnosti Microsoft, za zapůjčení svých produktů vysoké kvality po dobu mého studia zdarma. Bez těchto produktů by vznik této práce byl opravdu obtížný. Další společností, která si zaslouží poděkování je NVidia, která má dokumentace ke svým produktům, zejména pak frameworku PhysX, zpracovány opravdu kvalitně. Nakonec děkuji mé katedře za zapůjčení hardwarových prostředků, díky kterým jsem mohl na této práci pracovat.

Abstrakt

Náplní této diplomové práce je návrh a implementace simulátoru fotbalu robotů simulujícím reálnou fyziku. V první části se podívám na fotbal robotů obecně, abych vyvodil požadavky, které se na takovýto simulátor kladou. Poté se zaměřím na technologie, které ve svém řešení využívám k uspokojení těchto požadavků. Až Vás hrubě seznámím s použitými softwarovými prostředky, přistoupím k popisu vlastního řešení a seznámím Vás s jeho možnostmi a způsoby ovládání.

Klíčová slova: simulace, vizualizace, fotbal robotů, PhysX, OpenGL, FIRA

Abstract

The main goal of this diploma thesis is to design and implement a robot's soccer simulator capable of real physics simulation. In first part of this document, I will guide you through general principes of robot's soccer. Main purpose of this is to deduce requirements which applies on such simulator. Then i will focus on technologies, which i am using to comply with defined requirements. After you will get introduced with used software instruments, I will describe my solution its options and ways to control it.

Keywords: simulation, visualization, robot's soccer, PhysX, OpenGL, FIRA

Seznam použitých zkratek a symbolů

PhysX	–	NVidia PhysX SDK
Cooking	–	NVidia Cooking SDK
obj	–	soubor obsahující 3D model ve formátu Wavefront
TCP/IP	–	Transmission Control Protocol / Internet Protocol
MSDNAA	–	Microsoft Developer Network Academic Alliance
FIRA	–	Federation of International Robot-soccer Association
GLSL	–	OpenGL Shading Language

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V posledních letech se stal fotbal robotů zajímavou vědeckou disciplínou, ve které je nutno řešit řadu problémů z různých disciplín. Jako příklady bych uvedl oblasti umělé inteligence, zpracování obrazu a řídící a regulační techniky.

Jak je tomu ve fotbalu skutečném, tak i ve fotbalu robotů hrají dva mužstva robotů proti sobě podle definovaných pravidel. Tyto pravidla jsou pochopitelně přizpůsobeny technickým možnostem robotů.

Robotů, kteří hrají fotbal, je více druhů. Po světě existuje několik sdružení, které se právě stanovením technických parametrů a pravidel robotů zabývají. Jednou z těchto organizací je FIRA (Federation of Inernational Robotsoccer Association), která pochází z Jižní Koreje a byla založena roku 1995. Tato organizace definovala hned několik druhů robofotbalistů.

Všichni tito roboti však mají jednu společnou vlastnost, a tou jsou náklady na jejich pořízení a provoz. Cílem této práce je odstranit právě tuto překážku v technickém pokroku a umožnit tak rozšířit řady badatelů, případně umožnit těm stávajícím pracovat s roboty z pohodlí domova.

Prostředkem, kterým toho hodlám dosáhnout je implementace softwarového díla, simulátoru fotbalů robotů, který bude věrně simulovat chování robotů po stránce zákonů fyziky a rovněž bude poskytovat grafický výstup podobný skutečnosti. Navržený simulátor bude schopen simulovat všechny druhy robofotbalistů, kteří se svými vlastnostmi podobají robotům MiroSot definovaným sdružením FIRA.

2 Fotbal robotů

Účelem tohoto oddílu není plně zadefinovat fotbal robotů. To je mimo rámec tohoto textu, nicméně Vás má hrubě seznámit s vlastnostmi, které se budou v implementaci simulátoru uvažovat. Přesná pravidla naleznete na stránkách sdružení FIRA.

Klasický zápas fotbalistů MiroSot je řízen počítačem, jde o zápas počítač vs počítač, kde každý tým má pro pokyny svým robo fotbalistům, implementován svůj software. Tento software musí být schopen rozpoznat v jakém stavu se hra nachází a na základě toho rozhodnout jaké pokyny vyslat svým robotům. Vlastní roboti nemají žádný druh senzorů, takže naslepo poslouchají pokyny, které jsou jim bezdrátově zaslány z řídicí stanice. V kvalitě řídícího softwaru leží těžiště celé hry.

Aby bylo možno v řídicím počítači rozhodnout, kterému fotbalistovi zaslat jaká data, je plocha s fotbalisty snímána kamerou, a na základě získaného obrazu jsou vyhodnoceny pozice hráčů a jejich následující cíle. Tato řídicí smyčka se děje velmi rychle, takže se pohyb jeví plynulý.

Jednotliví hráči se od sebe odlišují za pomoci „dresů“. Jde o čtverec, upevněný na vrcholu robota, tak aby byl dobře vidět, který je barevně rozdělen do čtyřech částí. Dvě části jsou černé, jedna je obarvena barvou týmu a jedna má doplnující barvu. Barvy týmu jsou modrá, nebo žlutá. Doplňující barva určuje konkrétního robota, nebo jeho roli (brankář, obránce, záložník, útočník).

Vlastní hřiště pak je černé a matné, aby nedocházelo k nežádoucím odleskům. Počet hráčů a rozměry hřiště se mohou lišit podle druhu fotbalové ligy. FIRA definovala pro MiroSot dvě ligy. Střední, kde hraje za každý tým 5robotů na hřišti o rozměrech 220x180cm a branka je široká 40cm a Velkou, kde je rozměr hrací plochy 280x220cm a branka je široká 60cm. Hloubka branky je 15cm.

Roboty MiroSot si lze představit jako krychli s pravidly pevně stanovenými rozměry do maximálních 75x75x75mm. A maximální hmotností do 600g. Poháněni jsou dvěma koly. Stabilitu zajišťují stabilizační kuličky nebo válečky. Roboti jsou vybaveni rádiovým přijímačem pro příjem řídících příkazů, motorickým zařízením, schopným ovládat otáčky obou hnacích kol nezávisle na sobě a akumulátorem, který vydrží přibližně 10 minut. Tomu odpovídá hrací doba dvakrát 7.5minuty.

Hraje se s oranžovým golfovým míčkem o váze 46g a průměru 42,7mm. Před hrou jsou roboti rozmístěni ručně na své startovní pozice. Od spuštění hry nesmí být do jejího přerušení manipulováno jak s roboty, tak ani s řídícím počítačem. Při brance jsou roboti opět ručně rozmístěni.

Požadavky na simulátor vycházejí přímo z principu hry a z pravidel, kterými se hra řídí. Aby bylo možné vyvíjet software v oblasti zpracování obrazu, je nutné aby simulátor generoval grafický výstup podobný výstupu kamery, která by reálné roboty snímala. Tento výstup musí být rychlý, jako bychom skutečně snímali hru robotů a řízení mohlo být plynulé. Řekněme minimálně 25 snímku za vteřinu.

Pro vývoj herních strategií by pak bylo vhodné tento mezistupeň přeskočit a udávat pozice robotů přesně. Čímž nebude vývojový tým pro řešení umělé inteligence robotů zatížen chybou kolegů z oblasti zpracování obrazu a navíc se nebude muset s použitými technikami seznamovat a to ani na uživatelské úrovni.

Pro oblast řízení je pak zajímavé, aby roboti reagovali co nejvíce věrně reálným robotům. Aby se uplatňovaly známé zákony fyziky a aby se pak programy navržené s využitím tohoto simulátoru daly přenést na skutečné roboty fotbalisty.

Pokud si tedy požadavky shrneme, požadujeme výpočet fyziky kolem robofotbalu v reálném čase s možností výstupu grafického, tak i numerického.

3 Použité technologie

3.1 Přehled

Protože požadavky na simulátor jsou poměrně vysoké, a z programátorského pohledu i náročné, bylo potřeba se podívat po existujících technologiích, který by implementaci celého systému zjednodušily.

Prvním a zásadním požadavkem byla simulace reálné fyziky v reálném čase. Tento požadavek může uspokojit nasazení masivního paralelismu při výpočet fyziky díky technologii CUDA. Je známo, že na grafických kartách, bývá mnoho procesorů, které jsou velmi výkonné, nicméně mají podstatně užší sadu instrukcí než u hlavního CPU. Protože programování reálné fyziky samo o sobě je komplexní problém a navíc by programátora ještě omezovala i instrukční sada grafických procesorů, přišla společnost NVidia s frameworkem PhysX, který tuto úlohu řeší na mnohem vyšší programátorské úrovni. A já se rozhodl PhysX při své implementaci využít. Je důležité podotknout že tato technologie zatím není dostupná u ostatních výrobců grafických karet, takže pro běh simulátoru bude potřeba hardware s podporou PhysX.

Požadavky na vizualizaci se pak budu snažit naplnit pomocí dnes již standardních knihoven OpenGL, které jsou podporovány všemi grafickými kartami současnosti a jsou zdarma. Abych se graficky přiblížil pokud možno co nejvíce simulované realitě, rozhodl jsem se využít ve vizualizaci i shadery. Pro vývoj shaderů v jazyce GLSL pak poslouží vývojové prostředí Render Monkey od společnosti AMD. Také bude potřeba vytvořit modely robotů, kde se uplatní skvělý program Blender, který umí exportovat vytvořené modely do široce rozšířeného formátu Wavefront (dále jen .obj).

Základní aplikace pak je vystavěna na programovacím jazyce C++ s použitím knihoven Qt pro grafické uživatelské rozhraní. Těmto knihovnám se v textu nebudu dále věnovat, takže zájemce odkážu na [11]. Volba jazyka C++ vychází ze vztahu blízkosti knihovnám OpenGL, frameworku NVidia PhysX a také jazyku GLSL pro shadery. Také představuje rozumný kompromis mezi výkonností a blahem programátora. Zde bych chtěl rovněž vyzdvihnout existenci výborných vývojových prostředků společnosti Microsoft, které mám zapůjčeny v rámci MSDNAA po dobu studia zdarma.

3.2 NVidia PhysX SDK

3.2.1 Přehled

Nvidia PhysX SDK je framework umožnující simulovat fyziku pevných těles (rigid body dynamics) včetně detekce kolizí s vysokou věrohodností. Cílem této části není úplný vyčerpávající popis frameworku a jeho možností. Nicméně po přečtení by si měl čtenář udělat hrubou představu o tom, jakým způsobem PhysX pracuje.

Framework sám o sobě nabízí několik základních tvarů. Například kouli, kvádr, rovinu, přímku. Tyto tvary lze libovolně skládat pro vytvoření komplexního objektu - aktora. PhysX umožnuje pro každý tvar jednotlivě i pro celého aktora globálně nastavit rozložení hmotnosti, pozici, lineární rychlost, rotační rychlost, setrvačnost, tření a další fyzikální vlastnosti. Vlastnosti, které se dají nastavit se můžou lišit v závislosti na konfigurovaném

objektu. Díky tomu je uživatel schopen namodelovat mnoho typů reálných mechanických zařízení.

Jakmile vytvoříme pro každý objekt požadované simulace odpovídající abstrakce, můžeme je umístit do scény. Scéna sama o sobě umožňuje nastavení základních věcí jako je hustota prostředí, gravitace a podobně.

Mezi jednotlivé aktory scény pak lze modelovat vazby (constraints, joints), které mezi sebou aktoři mají. Příkladem může být třeba řetěz, který se skládá z více jednotlivých článků. Ty by byly namodelovány každý jedním aktorem. Je logické, že každý článek má vůči svému sousedu určité omezení, a to na rozdíl vzdálenostní. Znamená to, že pokud se jeden článek pohne, způsobí tím i pohyb svých sousedů z to důvodů zachování soudržnosti. Působí-li tedy na jeden článek síla, bude se přenášet i na další články řetezu rekurzivně.

Scéna sdružuje všechny objekty simulovaného světa a vazby mezi nimi. Framework zajistí, aby byly během simulace ve scéně dodrženy zákony fyziky a sada všech definovaných omezení. Celá simulace probíhá po časových krocích a ovládá se pomocí síly. Jinými slovy před každým krokem simulace je potřeba scéně říci, jak velké externí síly na simulované objekty působí, kde působí, kterým směrem a jak dlouhý časový úsek si přejete odsimulovat. Tyto síly se pak připočtou k těm, které scéna momentálně má. A dle fyzikálních zákonů se dopočte odpovídající stav scény, který nastane po aplikování sil za požadovaný časový úsek. Scén může být na kartě simulováno několik paralelně a krokovány mohou být nezávisle na sobě. Volání které způsobí výpočet simulace není blokující a lze během času, nutného k odsimulování provádět jiné výpočty.

K tomu všemu jsou data každé scény double buffrovány, takže lze pracovat s daty scén, které jsou momentálně simulovány. Pochopitelně budou k dispozici data, která byla platná před odstartováním simulace. Toho lze využít například k vizualizaci scény během výpočtu nového stavu. Pokud bude frekvence simulačních kroku dostatečně vysoká, je rozdíl mezi simulovanou a vizualizovánou scénou zanedbatelný.

Kolize ve scéně jsou rovněž řešeny frameworkem, zákon o neprostupnosti hmoty je brán při simulování v potaz. Tedy kromě speciálních případů, ke kterým se vrátíme později v podkapitole věnované detekci kolizí, se budou aktoři od sebe odrážet.

Kromě výše uvedených vlastností nabízí PhysX také další funkcionalitu, jako fyziku látek (cloth). Fyziku kapalin a plynů, Měkkých těles (to jsou ta ohebná, která se mohou deformovat během simulace), silová pole (simulace tornád, gravitací) a mnoho dalších. Nicméně z důvodu rozsáhlosti frameworku se v příštím omezím pouze na stručný popis funkcí, které jsem při implementaci simulátoru využil. K frameworku PhysX jsem se během své práce nedostal k žádné tištěné literatuře. Důvod je prostý, nebylo to potřeba. Dokumentace k frameworku PhysX, kterou NVidia ke svému produktu dodává, je na špičkové úrovni. Součástí jsou rovněž ukázkové aplikace, zdrojové kódy a učební lekce krok za krokem. Vše v základní distribuci knihoven. Proto hlubšího zájemce o tuto technologii odkazuji přímo na dokumentaci dodanou k produktu.

3.2.2 Základy API

- Architektura
  - PhysX SDK má programátorské rozhraní dle ANSI C++. Vnitřně je implementován jako hierarchie tříd. Každá třída, která nabízí uživatelskou funkcionalitu implementuje rozhraní. Toto rozhraní je abstraktní třídou C++. Navíc jsou vystaveny některé funkce, které lze volat staticky.
- Konvence
  - Všechny třídy jsou definovány v hlavičkovém souboru stejného názvu, jako vlastní třída
  - Datové typy a jména tříd začínají velkým písmenem
  - Všechny třídy rozhraní (třídy s uživatelskou funkcionalitou) začínají „Nx“
  - Funkce a proměnné začínají malým písmenem
  - Návratové hodnoty a parametry funkcí, kde je přijatelná i hodnota NULL jsou kódovány s hvězdičkou (* pointer) a tak je potřeba ověřovat návratovou hodnotu
  - Návratové hodnoty a parametry funkcí, kde hodnota NULL přijatelná není, jsou kódovány s uppersand (& reference).
- Datové typy.
  - Aby byla zajištěna určitá přenositelnost[1] využívá PhysX vlastní datové typy definované v NxSimpleTypes.h. Namátkou třídy NxVec3, NxMat33, NxQuat reprezentují trojrozměrný vektor floatů, matici 3x3 prvků nebo quaternion. Součástí PhysX jsou i třídy, zabývající se matematikou pro snadnou převoditelnost těchto datových typů. K volání funkcí PhysX se však vyžaduí typy s prefixem Nx.
- Jednotky
  - PhysX nepotřebuje pro svou práci použití žádné konkrétní soustavy jednotek. Nicméně je důležité zavést si soukromou konvenci, aby se předešlo neočekávaným výsledkům. SDK pro svou práci využívá tři typy jednotek. Je třeba měřit hmotnost, vzdálenost a čas. Všechny ostatní jednotky jsou pak logicky odvozeny z těchto tří. Např. Rychlost je vždy podíl vzdálenosti a času. Protože PhysX pracuje pouze s 32bit čísly, je vhodné při návrhu používaných základních jednotek brát přesnost floatu v potaz. A použít jednotky takovým způsobem, aby se dosáhlo požadované přesnosti.

[1] PhysX je multiplatformní

3.2.3 Tvary (Shapes)

Jak již bylo zmíněno výše. Tvary jsou základními stavebními bloky používanými ke konstrukci aktorů (avatarů skutečných simulovaných objektů) ve scéně. V PhysX SDK se tvary dělí na základní a meshe. Základní tvary mohou existovat samy o sobě, PhysX je zná, dokáže je simulovat. Meshe jsou tvary, které je teprve potřeba pro PhysX vytvořit. Jinak jsou jen prázdnými šablonami.

Základní typy, které PhysX pro modelování reality nabízí jsou:

NxBounds3 - Jedná se o osově orientovaný bounding box. V podstatě kvádr, který však nepodporuje rotace.

NxBox - Jedná se o kvádr. Rozměr se zadává jako vzdálenost rovin od středu. Krychli o rozměrech x,x,x namodelujete voláním konstruktoru s parametry x/2,x/2,x/2. Vnitřní prostor není vyplněn.

NxCapsule - „Pilulka“. Jde o ekvidistantu k úsečce. Parametry konstruktoru jsou úsečka a vzdálenost ekvidistanty. Vnitřní prostor je vyplněn.

NxPlane - Rovina. Konstruktor je určen vektorem normály a vzdáleností od středu. Pozor, není určena k modelování stěn. Strana roviny opačná uvedené normále je jakoby vyplněná a všechny tělesa se budou snažit tento poloprostor opustit. Toto chování je způsobeno technikou vyhodnocování kolizí s rovinou.

NxRay - Přímka. Jednoznačně určena dvěma body.

NxSegment - Úsečka. Proti přímce končí v definovaných bodech.

NxSphere - Koule. Definována bodem a poloměrem. Vnitřní prostor je vyplněn.

3.2.4 Meshe

Meshe nejsou základními tvary, a nelze je vytvořit pouhým voláním konstruktoru. Postupy vytváření meshů a omezení na ně kladená se liší podle jednotlivých typů. Ke konstrukci se využívá Cooking SDK, dodávané spolu s PhysXem. Různé typy meshů se hodí pro různé aspekty simulace. Je třeba podotknout, že né všechny typy meshů mají připraveno řešení kolizí se všemi ostatními typy. Tomu se podrobněji budeme věnovat v kapitole řešení kolizí.

NxConvexShape - Jedná se o mesh, který vznikne pomocí definování vrcholů konvexního tělesa. Maximální možný počet těchto bodů je 256 na jeden shape. Tento mesh jako jediný nabízí řešení kolizí proti všem ostatním tvarům v PhysX. Vhodný pro modelování aktorů, které nelze vymodelovat za použití základních tvarů

NxTriangleMeshShape - Jde o mesh, který vznikne pomocí definování vrcholů a indexů na tyto vrcholy. Normály se explicitně neuvádějí a výpočet probíhá (v1-v0) x (v2-v0).

Toto je nutné brát při konstrukci na zřetel, protože detekce kolizí pracuje pouze ve směru definovaných normál. Vhodný pro modelování terénu.

NxHeightFieldShape - Vhodný pro definování terénu. Tento mesh vznikne pomocí definování pole vzorků výšek. Díky pravidlům, které platí pro výškové mapy je výkonnější než obecný NxTriangleMesh, nicméně nelze modelovat všechny typy povrchů. (Například převisy)

Speciálním tvarem pak je:

NxWheelShape - Který nepatří mezi základní tvary, ale ani mezi meshe. Jeho úkolem je simulovat kolo. Kromě tvarové reprezentace lze simulovat i parametry podvozku, pneumatik a síly motorických zařízení. Tomuto tvaru se budeme věnovat podobněji v kapitole věnované modelování robotů.

3.2.5 Aktoři (Actors)

Aktoři jsou základním kamenem simulace. Jde o reprezentaci skutečných objektů reálného světa zjednodušených tak, aby stále vystihovali fyzikální podstatu věci. Aktoři vznikají skládáním základních tvarů a modelováním jejich vlastností. V PhysX existují dva základní typy aktorů - statičtí a dynamičtí.

Statický aktor je pevně svázán se svou pozicí ve scéně. Modelují se s ním nerozbitné a nepřesunutelné objekty. Jejich primárním úkolem v simulaci je detekce kolizí. Takže by měli mít vždy definované nějaké tvary.

Dynamický aktor pak slouží k simulování objektů scény, které jsou ovlivňovány silami. Může jít klidně o abstraktní bod s hmotností, připojený přes vazbu (joint) k jinému aktoru scény (např. simulace kyvadla hodin). Proto není bezpodmínečně nutné mít u dynamického aktora definovány tvary.

Podle zákonů fyziky, jde každý pevný objekt libovolného tvaru reprezentovat jeho momentem setrvačnosti (inertia tensor) a hmotností v bodě jeho těžiště. Toto je i přístup frameworku PhysX. Pokud definujeme aktorovi tvary, PhysX tyto tvary pro účely simulace stejně převede. Stejně jako u statických aktorů však má definování tvarů zásadní význam pro detekci kolizí.

Při konstrukci se tyto dva druhy aktérů liší povinností nastavit parametry body. Pokud nenjsou parametry body nastaveny, jde o aktéra statického. Parametry body lze dopočítat ze tvarů aktora automaticky, nebo lze zadat hmotnost a moment setrvačnosti, a pak se parametry jednotlivých tvarů ignorují. Pokud hmotnost neznáte, lze také nastavit jen hustoty materiálu pro jednotlivé tvary, a hmotnost se dopočte. Tato funkcionalita může být užitečná v případě, že se parametry simulace s časem mění. Bohužel tyto údaje nemusí souhlasit v případě že se tvary aktora přes sebe překrývají, protože v reálném světě by byl jeden z tvarů o část svého objemu ošizen. Při automatickém stanovování objemu odvozeného z hustot použitých materiálů však v místech překryvu dochází k sčítání hustot. Pokud je aktér vyroben z jednoho materiálu, má tento automatický výpočet smysl, pokud materiály kombinujete, je stanovení hustoty pochopitelně složitější, než objekt zvážit. Zadávat však hustotu místo přímé hmotnosti aktora však má zásadní smysl

pro použití v simulátoru, protože lze automaticky dopočítat novou hmotnost aktora po změnách parametrů simulované reality přímo frameworkem PhysX na grafické kartě. Uvolněný procesorový čas pak lze využít pro časově náročné opravy vizuálních modelů.

3.2.6 Vazby (Joints)

O vazbách se zmíním jen rámcově. Protože finální verze simulátoru již explicitně vazeb nevyužívá. Pochopitelně jsou vazby využívány interně frameworkem. Vazby poskytují způsob, jakým propojit dva aktory. Vazba pak po simulaci vyžaduje, aby se tito dva aktéři vždy hýbali podle pohybu partnera. Omezuje se jejich relativní pohyb. Druh omezení pohybu záleží na typu vazby. Může jít o pohyb rotační, lineární nebo kombinace obojího.

Cena[2] simulování vazby závisí na jejím stupni volnosti. Stupňů volnosti máme 6. Tři pro lineární pohyb po osách x,y,z a tři pro pohyb rotační kolem zmíněných os. Čím více stupňů volnosti páru těles odejmeme, tím více je simulace vazby náročná.

U již dříve zmíněného případu simulace řetězu by se mezi sousedními články řetězu vytvořily vazby, omezující relativní pohyb mezi jednotlivými články po osách x,y,z. Rotační pohyb by se neomezoval. (NxSphericalJoint) V případě opravdu přesné simulace by se omezoval i rotační pohyb na maximální úhly, kterých lze při ohybu mezi články fyzicky dosáhnout. (NxD6Joint)

Pomocí PhysX lze modelovat i vazby rozpadnutelné. Tedy že určitá síla, která působí na vazbu, jí přeruší, čímž lze simulovat například realitu přetržení řetězu. Které vazby PhysX nabízí a jejich podrobný popis lze nalézt v dokumentaci. Navíc jsou součástí distribuce PhysXu i úkázkové programy využívající různých vazeb, kde jde hned vidět, co která vazba dělá.

3.2.7 Řešení kolizí

Základy

Kolize jsou ve většině případů řešeny nativně přímo v frameworku PhysX. Prvním krokem řešení kolizí je zjištění které objekty scény vůbec mají šanci se dotýkat. Protože ale při n tvarech ve scéně existuje zhruba n*n/2 potencionálních párů, které se mohou dotýkat, ověřování všech možností by bylo časově příliš náročné. Místo toho SDK automaticky rozdělí prostor na podprostory. Tímto způsobem jsou pak na kontakt testovány pouze blízké tvary. Obecně řešení kolizí může dojít až k velmi komplexním algoritmům. Důkazem toho je i publikace [6] na kterou případné zájemce o tuto problematiku odkazuji.

Kolizní skupiny

Objekty je možné přiřazovat do kolizních skupin. Celkem je ve PhysXu připraveno 32 skupin. Pokud skupinu nepřiřadíte, je aktor zařazen do implicitní skupiny 0. Kolize jsou vždy vyšetřovány jen v rámci nastavení matice groupCollisionFlag, která určuje, které skupiny mezi sebou kolidují a které ne. Implicitní nastavení této matice jsou samé

[2] ve smyslu výpočetního času

jedničky, tedy kontakt každý s každým. Pro simulaci reálných scén je toto implicitní nastavení žádoucí.

Triggery

Na bázi detekce kolizí pracují také triggery. Trigger je tvar, který dovoluje jiným tělesům pronikat skrze své hranice. Každý simulovaný krok pak vytváří událost pro každý tvar, který je uvnitř jeho objemu. Existují tři typy generovaných událostí. Když do objemu triggeru nějaké těleso vstoupí, pokud vystoupí a nebo pokud v něm zůstává. Triggery pak můžou sloužit pro naprogramování herních událostí na základě pozicí objektů scény. Ve fotbalu robotů je například využívám pro detekci míčku za brankovou čárou. Kdy je potřeba znovu rozmístit roboty. Ke scéně, která využívá triggery je pak nutné registrovat obslužnou třídu pro zpracování událostí vygenerovaných triggery ve scéně. Obslužná třída musí implementovat rozhraní NxUserTriggerReport.

Spojitá detekce kolizí

Spojitá detekce kolizí řeší problémy, kde klasická detekce kolizí selhává. U běžné detekce kolizí můžou rychle se pohybující objekty projít skrze jiný tvar v čase mezi jednotlivými simulačními kroky, a tak na ně nemůžou být aplikovány síly, které by jinak zajistily odraz, nebo zastavení takového průniku. Abychom tomuto problému dokázali čelit, je PhysX vybaven možností, netestovat pouze pozice aktorů v bodech simulace, ale testovat celý objem, který aktoři urazí během simulované doby. Využití této techniky je sice časově náročnější, ale v některých situacích jde o jediné řešení, které zachovává realističnost simulace. Je důležité podotknout, že se stále jedná o akcelerované řešení a mnohanásobně překonává výkonnost vlastních pokusů o nápravu těchto průniku na procesoru.

Vzájemná detekce kolizí u tvarů

Detekce kolizí je automaticky řešena u všech základních tvarů. Také lze aplikovat spojitou detekci kolizí. Problém nastává u meshů, kde algoritmus řešení kolizí pro obecný mesh může být časově velmi náročný a proto jej PhysX nepodporuje. Plná podpora detekce kolizí je implementována jen u tvaru NxConvexMesh. Tento tvar jako jediný mesh dokáže kolidovat se všemi ostatními tvary, které PhysX nabízí. Podpora kolizí u ostatních tvarů je zobrazena v tabulce 1.

Pokud vyžaduje simulovaná realita řešení kolizí u složitějších než základních tvarů, jsou tu dvě možnosti. Implementovat si vlastní způsob řešení kolizí, nebo poskládat Vámi požadovaný tvar z více tvarů NxConvexMesh.

Raycasting

Posledním prvkem frameworku PhysX, kterému se budeme věnovat je raycasting. Svým způsobem se může jednat o nástroj řešení kolizí. Nicméně jej lze využít třeba i pro akceleraci vykreslování metodou raytraycing, výběru objektu pomocí myši nebo jiných úloh, kde je potřeba pracovat s paprsky.

	Sphere	Box	Capsule	Plain	Convex	Heightfield	Wheel	TriangleMesh	Heightfield	Pmap
Sphere	ano	ano	ano	ano	ano	ano	ano	ano	ano	ano
Box	ano	ano	ano	ano	ano	ano	ano	ano	ano	ano
Capsule	ano	ano	ano	ano	ano	ano	ano	ano	ano	ano
Plane	ano	ano	ano	ne	ano	ne	ano	ano	ne	ne
Convex	ano	ano	ano	ano	ano	ano	ano	ano	ano	ano
Heightfield	ano	ano	ano	ne	ano	ne	ano	ne	ne	ne
Wheel	ano	ano	ano	ano	ano	ne	ano	ano	ano	ano
TriangleMesh	ano	ano	ano	ano	ano	ne	ano	ne	ne	ne
Heightfield[a]	ano	ano	ano	ne	ano	ne	ano	ne	ne	ne
Pmap[b]	ano	ano	ano	ne	ano	ne	ano	ne	ne	ano

Tabulka 1: Tabulka frameworkem podporovaných kolizí

[a] Jde o TriangleMesh s nastaveým flagem heightmap
[b] Jde o TriangleMesh s nastaveným flagem pmap

Raycasting umožňuje vyslat do scény paprsek. PhysX pak dopočítá, které objekty byly paprskem zasaženy a na které souřadnici. Jaká byla normála k tělesu v bodě dopadu a podobně. Podle účelu, kterému bude námi vyslaný paprsek sloužit, lze provádět různě časově náročné sledování paprsku. V podstatě máme šest možností. Kdy polovina volání pracuje se skutečnými tvary aktorů a druhá polovina pouze s jejich automaticky stanovenými bounding boxy.

Nejprimitivnější vyslání paprsku je raycastAnyBounds a raycastAnyShape, kde je výsledkem volání jeden zásah, není však zaručeno že jde o zásah neblíže zdrojovému bodu paprsku. Komplexnějším voláním je pak raycastClosestBounds, raycastClosestShape, kde je zajištěno, že vrácený zásah bude nejbližším zásahem v definované scéně.

Nakonec PhysX nabízí volání raycastAllBounds, či raycastAllShapes, které vracejí všechny průsečíky paprsku s objekty ve scéně. Protože počet vracených výsledků se může značně lišit, je při každém nalezeném průsečíku generována událost jejíž součástí je jeden průsečík. Podobně jako u triggerů je i zde potřeba se k odběru událostí z konkrétní scény zaregistrovat. Toto volání raycastingu je pak logicky časově nejnáročnější. Třída zpracující události musí implementovat rozhraní NxUserTriggerReport.

3.3 OpenGL

3.3.1 Úvod

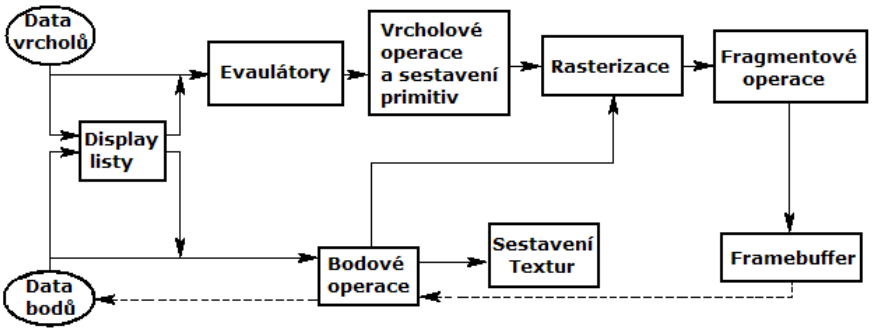
OpenGL je knihovnou sloužící k vykreslování 2D a 3D grafiky a vznikla počátkem devadesátých let díky spolčnosti Silicon Graphics. V dnešní době jde o široce rozšířený a uznávaný průmyslový standard. Protože o OpenGL existuje řada výborných publikací, ze kterých můžu doporučit především [1], rozhodl jsem se tuto kapitolu pojmout velmi stručně.

OpenGL je programovacím rozhraním k funkcionalitě grafické karty. Toto rozhraní se skládá přibližně ze 150-ti různých příkazů[3], díky kterým je možno definovat objekty a operace, které jsou potřeba pro interaktivní 3D aplikace. OpenGL samotné neumožňuje vykreslování složitých geometrických útvarů. Pro zajištení široké hardwarové podpory a implementací na více platformách, jsou k dispozici pouze jednoduché primitivy, jakými jsou body, čáry, trojúhelníky a polygony. Případně jejich stripy. Způsobů skládání je celá řada a existují techniky, které umožní významně rychleji vykreslit výsledný obraz, pokud jsou dodrženy. Techniky akcelerací je možné nastudovat například z knihy [7].

Spojováním těchto primitiv je možné vytvořit objekt libovolného[4] tvaru. Pro usnadnění programátorské práce, jsou nad knihovnou OpenGL vybudovány i další sofistikovanější knihovny, které umožňují kreslit složitější primitivy. Součástí každé implementace OpenGL bývá dnes i knihovna GLU, která obsahuje například kouli, kvádr, kvadratické povrchy nebo křivky NURBS.

[3] dle normy ANSI C

[4] pochopitelně zde nějaký malý rozdíl bude, protože ve finále OpenGL kreslí trojúhelníky



Obrázek 1: Pipeline OpenGL

3.3.2 Pipeline

Ve většině implementací je stejná, nebo velmi podobná sekvence kroků, která vede k finálnímu vykreslení obrázku na zobrazovacím zařízení. Tomuto pořadí zpracování říkáme pipeline. Do značné míry je spjata a odvozena ze skutečného hardware, který dokázal vykonávat jednotlivé operace. V dnešní době je architektura grafických akcelerátoru dále, a jednotky jsou programovatelné tak, aby mohly vykonávat více činností. Nicméně architektura OpenGL částečně tuto historickou skutečnost odráží. Pipeline OpenGL si můžete prohlédnout na obrázku 1.

3.3.3 Způsob programování

První věcí, se kterou se musíme alespoň letmo seznámit, je syntaxe příkazů. Každý příkaz, který je součástí knihovny začíná malým gl. (např. glVertex3f()). Konstanty definované v OpenGL jsou psány celé velkými písmeny. (např. GL_LIGHT0). U příkazů si můžeme povšimnout „podivné“ koncovky. Ta nás informuje o typu vstupních dat, které funkce očekává. V případě glVertex3f jsou jako parametry očekávány tři čísla typu float. Mnoho příkazů se liší právě jen svou koncovkou, což znamená, že je lze volat s různými vstupními parametry. Další koncovka, která se může ve jméně příkazů vyskytnout je v. Například glVertex3fv(). Tato koncovka nás informuje, že se očekává pouze ukazatel na vektor (pole). V případě příkladu se očekává pole o třech prvcích typu float. Některé funkce umožňují

volání s oběmi varinantami. Některé lze volat pouze s předáním ukazatele na vektor, některé zase pouze s přímo vloženými hodnotami. Vyčerpávající popis všech funkcí a jejich syntaxe však je daleko mimo rozsah tohoto textu, proto Vás jen odkážu na [3] nebo [1].

Knihovna OpenGL, která svými příkazy zastupuje grafickou kartu, je v podstatě stavový stroj. Pokud nějaký parametr změníme, zůstane změněn tak dlouho, než jej opět jiným voláním přepíšeme. Toto chování je ve většině případů velmi žádoucí. Chceme-li kreslit objekty jedné barvy, nastavíme první barvu a pak můžeme volat příkazy, které tvoří námi požadované geometrie.

Po inicializaci je kamera, která je vašim pohledem do vytořené scény, umístěna ve středu souřadné soustavy a zarotována ve směru osy -z. Osa X jde doprava, osa Y směřuje nahoru. Toto implicitní nastavení však neumožnuje kreslení bez hlubší znalosti funkce OpenGL. Protože zobrazovaný prostor, který kamera nabízí je před prostorem zobrazovým monitorem. K nastavení správné pozice kamery poslouží jedna z funkcí gluOrtho2D(), glOrtho(), gluPerspective() nebo gluLookAt(), kde poslední jmenovaná je uživatelsky nejvíce příjemná. Pokud Vám postačuje statická kamera, stačí tuto funkcni zavolat pouze jednou v metodě pro inicializaci. Vlastní kresba pak probíhá v metodě draw()[5]. V této metodě musíte zavolat takové příkazy, aby plně vykreslili vámi požadovaný obraz. Obvyklá struktůra této metody bývá vymazat starý obsah obrazovky a nakreslit požadované geometrie. Často se z důvodu lepšího výsledného dojmu používá metoda doublebufferování, čímž je zajištěno, že je vždy zobrazen celý obrázek a obraz neproblikává. Pak v závěru metody draw() bude navíc také volání, které vymění obraz, který jste právě nakreslili za ten, který byl do té doby zobrazován. Jednoduchou funkci, která nakreslí čtverec si lze prohlédnout ve výpise zdrojového kódu 1.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glBegin(GL_POLYGON);
    glVertex3f (−1, 1, 0.0);
    glVertex3f ( 1, 1, 0.0);
    glVertex3f ( 1, −1, 0.0);
    glVertex3f ( −1, −1, 0.0);
glEnd();
glFlush();
glutSwapBuffers();
```

Výpis 1: Ukázka velmi jednoduchého kódu OpenGL

V kódu se vyskytují ještě další důležité volání, které jsem opoměl zmínit. Jde o funkce glBegin() a glEnd(). Tyto funkce označují začátek a konec kreslené primitivy. Typ kreslené primitivy určuje vstupní konstanta ve volání glBegin(). Pokud se kreslí ku příkladu trojúhelník, očekávají se volání nakresli vrchol (glVertex) po trojicích. Z každých tří volaných vrcholů je pak sestrojen jeden trojúhelník. Pokud je vrcholů uvedeno méně, nekreslí se nic. Pokud více, pokusí se OpenGL nakreslit tolik instancí dané primitivy, na kolik mu bude stačit počet podaných vrcholů. Typy primitiv se tento text z prostorových důvodů nebude zabývat. Zájemci o OpenGL pak mohou vzít do ruky opět třeba skvělou [1].

[5]Jméno metody však nemusí být závazné, třeba ve widgetech vykreslujících 3D v knihovnách Qt bývá metoda pojmenována paintGL()

3.3.4 Evaulátory

Hladké povrchy se v OpenGL aproximují velkým množstvím jednodušších primitiv. Je však známo, že řada těchto povrchů a křivek se dá matematicky vyjádřit na základě několika parametrů, jako jsou řídící body. Ukládání několika řídicích bodů je paměťově mnohonásobně úspornější, než ukládat vlastní aproximující tvary. Navíc, takto uložené tvary jsou jen aproximací, zatímco uložení řídicích bodů popisuje skutečnou křivku, kterou lze později dopočíst v libovolné přesnosti a počtu aproximujících primitiv. K výpočtům takovýchto aproximujících primitiv slouží v OpenGL evaulátory. Na základě řídicích bodů (nebo dalších přídavných attributů třeba pro NURBS) pak evaulátory samy polohy vrcholů a jim odpovídající normály aproximujících primitiv dopočtou. Popisy křivek, které lze pomocí evaulátorů a různých nadstaveb OpenGL dopočíst lze nalézt rovněž v [1].

Pro účely seznámení se s funkcionalitou mého simulátoru však postačí, když Vás seznámím s postupem pro výpočet Beziérovy křivky zadané řídicíi body v poli. První krok, který musíme provést, je zadat evaulátorům řídicí body. K tomuto slouží volání glMap(). Variant tohoto volání je více, podle typu křivky (plochy), který hodláme kreslit. Pro kreslení křivky jde o 1 rozměrné evaulátory, pro výpočet ploch jde o dvojrozměrné evaulátory. Rozměrem se zde rozumí počet proměnných parametrů křivky v jejím matematickém popise. Beziérova křivka má jeden takový parametr. Použije se zde tedy glMap1fd() a to podle toho, s jakou precizností máme připraveny řídicí body[6].

Prvním parametrem je typ řídicích bodů. Pro evaulaci Beziérovy křivky v 3D prostoru půjde o 3D body, použije se tedy konstanta GL_MAP1_VERTEX3. Na tomto místě bych rád poukázal na to, že evaulátory lze použít i pro další výpočty, jako je například míchání barev. A právě proto je evaulátor napsán robustně, a akceptuje různé druhy vstupních dat. Dalším parametrem volání je pak startovní a konečná hodnota parametru. Pokud chceme celou křivku, uvedeme hodnoty 0 a 1. Pokud máme zájem vykreslovat jen část[7], můžeme tak učinit přávě na základě těchto parametrů. Dalším parametrem je počet floatů, které zastupují řídicí bod. Tato hodnota odpovída počtu souřadnic prostoru, pro který se výpočet provádí. Předposledím parametrem je je řád křivky. Tedy, kolik řídicích bodů se bere k výpočtu každého bodu křivky. Technika ke stanovení této hodnoty je stupeň křivky + 1. Tedy pokud jde o kubiku stupně 3, je potřeba hodnota 4. Posledním a tím nejdůležitějsím parametrem je pak ukazatel na řídicí body. Po zavolání této metody je možno aktivovat evaulátor. To provedeme voláním glEnable(GL_MAP1_VERTEX_3). V tento moment je evaulátor připraven pro výpočty. A jednotlivé body ležící na křivce pak získáváme voláním glEvalCoord1fdv(). Pro názornost připojuji ukázku vlastního kódu 2, kde se vypočítává křivka dráhy robota na základě řídicích bodů. Typ křivky je Beziér.

```
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &robot->ridici_body_trajektorie[0]);
glEnable(GL_MAP1_VERTEX_3);

glBegin(GL_POINTS);
for (int i = 0; i < POCET_BODU_TRAJEKTORIE; i++)
```

[6] f jako float, d jako double

[7] třeba pokud kreslíme křivku po obloucích

```
{
    glEvalCoord1f((GLfloat) i /( POCET_BODU_TRAJEKTORIE-1));
}
glEnd();
```

Výpis 2: Použití evaulátoru pro výpočet trajektorie robota

3.3.5 Feedback Mode

Protože opravdová 3D aplikace musí umět nejen scénu vykreslit, ale také umožnit s kreslenými objekty dále manipulovat, je OpenGL vybaveno třemi módy, ve kterých dokáže operovat. Primárním módem je pochopitelně vykreslování (render). Při inicializaci je OpenGL nastaveno právě do tohoto módu. Pro výběr objektů scény je OpenGL vybaveno mechanismem select, který automaticky řekne, který objekt je kreslen do konkrétní zájmové oblasti. Tento mechanismus se dá využít k implementaci kódu, který pak určí, na který objekt uživatel ukazuje. Třetím módem, kterým OpenGL disponuje, je mód feedback. Tento mód způsobuje, že místo aby byla data pro vykreslení zasílána do grafické karty a následně na výstupní zařízení, jsou uživateli vrácena zpět do předem nachystaného bufferu. Prvníma dvěma módy, se zde nebudeme zabývat. Pro pochopení funkcionality přídavné logiky simulátoru, která umožnuje řídit roboty tak, aby se vyhýbali kolizím, však bude zapotřebí alespoň hrubý nástin toho, jak pracuje mód feedback.

Jak jsme již řekli, data, která by se normálně kreslila, budou v feedback módu vráceny uživateli. Proto je logicky prvním krokem při jeho použití stanovit, kam data zapsat. K tomuto účelu slouží volání glFeedbackBuffer(). Parametry tohoto volání popisují jaky typ dat bude zapsán a kolik jich maximálně bude. Prvním parametrem je velikost bufferu ve floatech. Druhý parametr říká, jaká data budou uvnitř. A třetím parametrem je pak vlastní odkaz na předpřipravené pole. Druhý parametr zde hraje klíčovou roli, protože různá data vytvářejí různé struktury ve vlastním bufferu. Pro detaily odkazuji na [1], kde jsou všechny možnosti detailně popsány.

Po volání metody glFeedbackBuffer() můžeme konečně přepnout OpenGL do feedback módu. Učiníme tak voláním glRenderMode(), která slouží pro přepínámí mezi třemi módy, které OpenGL má. Volba módu, do kterého se chceme dostat se provádí jediným parametrem funkce, definovanou konstantou. V OpenGL jsou k tomuto účelu definovány tři konstanty: GL_RENDER,GL_SELECT a námi požadovaný GL_FEEDBACK.

Cokoli od teďkom vykreslíme, bude pouze přeposláno do předpřipraveného bufferu. V módu feedback můžeme pochopitelně využívat také evaulátory, což je pro funkcionalitu mé přídavné logiky klíčové a proto tuto skutečnost explicitně zmiňuji. Abychom se na zapsaná data mohli podívat, musíme opustit mód feedback. Právě při přepnutí módu se totiž shromážděná data kreslených primitiv překlopí do námi nachystaného pole. To učiníme opětovým voláním glRenderMode().

Mód do kterého aplikaci přepneme po feedbacku není rozhodující, nicméně důležitá je návratová hodnota, která nás informuje o počtu floatů, které byly do bufferu zapsány. Sledováním této hodnoty můžeme snadno zjistit chybu v naší aplikaci. Důležité na tomto místě je uvědomovat si, že kromě vlastních primitiv jsou data proložena ještě jakýmisi

informačními štítky, které říkají o jaký typ primitivy jde. Podle těchto hodnot se dá buffer později pohodlně parsovat. Pokud nám pro vlastní parsování základní štítky vkládané mezi primitivy nepostačují, můžeme si v procesu kreslení, kde vlastní data připravujeme, do dat přidávat vlastní značky a to pomocí funkce glPassThrough(). Tato funkce do feedback bufferu na odpovídajícím místě přidá dvě hodnoty float. Jedna je parametrem volání a druhá info token, tedy onen štítek, že jde o hodnotu předanou pomocí glPassThrough(). Ukázku použití feedback módu si můžete prohlédnout ve výpise zdrojového kódu 3.

```
// velikost bufferu feddbacku bude
//pocet_prvku * ( poc_bodu_na_prvek * pocet_souradnic + 1_token_info) + 2_passthrough
const unsigned int feedbackBufferSize = POCET_BODU_TRAJEKTORIE*(1*3+1)+ 2;
GLfloat feedBuffer[feedbackBufferSize];

// prirad buffer pro feedback mode
// obsahem budou x,y,z souradnice pro kazdy bod
glFeedbackBuffer(feedbackBufferSize, GL_3D, feedBuffer);

//prepni do feedback modu (nekresli, pouze pocitej data)
glRenderMode(GL_FEEDBACK);

glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &robot->ridici_body_trajektorie[0]);
glEnable(GL_MAP1_VERTEX_3);

glBegin(GL_POINTS);
for (int i = 0; i < POCET_BODU_TRAJEKTORIE; i++)
{
   glEvalCoord1f((GLfloat) i /( POCET_BODU_TRAJEKTORIE-1));
}
glEnd();
glPassThrough(6969);

// vrat se zpet do render modu = napln feedBuffer
int size = glRenderMode(GL_RENDER); //size ~ pocet floatu vracenych do bufferu
if (size != feedbackBufferSize){//chyba pri vypoctu}
```

Výpis 3: Použití feedback módu

3.3.6 Shadery

Současným trendem ve vývoji grafických karet je zobecnování použitých jednotek. Kdysi fixní jednotky, které byly navrženy pro jediný úkol v pipeline grafické karty s cílem vykreslení výsledného obrazu byly zaměňovány za nové, které mohly vykonávat takové činnosti, jakých je při vykreslování aktuálně nejvíce potřeba. Jejich univerzálnost však vedla k jejich čím dál větší návrhové složitosti a jejich výroba se stávala technologicky velmi obtížnou. Proto se zvolila jiná cesta.

Dnešní jednotky jsou programovatelné, a pracují podle toho, jaký program je v nich natažen. Limitujícím faktorem dneších karet je tak celkový počet těchto jednotek a jejich schopnost adaptivně měnit své role, aby byly stále vytíženy všechny a aby prováděly právě takovou činnost, která je pro daný typ vizualizační úlohy potřeba. Dnešní pro-

gramovatelné jednotky tak lze podle úloh dělit do dvou základních skupin. Jednotka, která počítá s daty geometrií (vertex processor) a jednotka která počítá s bodovými daty (fragment processor). Každý pár takových jednotek tak je schopen nahradit část pipeline OpenGL podle svého naprogramování. Tento přístup nejenže dovoluje nahradit veškerou funkcionalitu příslušné části pipeline OpenGL, ale také ji do značné míry rozšířit. Například o nové způsoby osvětlění, úpravy normál a podobně. Vše podle programu, podle kterého se jednotka řídí. Pro programování takovýchto jednotek slouží Shading language. Já osobně ve své práci používám OpenGL Shading Language (dále jen GLSL).

Jazyk GLSL je svou syntaxí odvozen z C. Má bohatou zásobu datových typů včetně matic nebo vektorů a je tak velmi vhodný pro 3D aplikace. Jazyk obsahuje podporu pro smyčky, volání subrutin a podmíněné výrazy. Je možné deklarovat proměnné uvnitř těla programu. Rozsáhlá skupina vestavěných funkcí zpřístupňuje schopnosti potřebné pro vykreslovací alogoritmy. Jde zkrátka o procedurální jazyk vyšší úrovně. Navíc jde stejně jako u OpenGL o multiplatformní a na operačním systému nezávislé programovací rozhraní pro 3D vizualizaci. Pokud vás tato upoutávka zaujala, zkuste se podívat do [2] nebo [8], které se problematikou shaderů podobně zabývají.

4 Simulátor

4.1 Přehled

Po důsledném zvážení požadavků a po prostudování technických prostředků, které jsem si pro řešení simulátoru fotbalu robotů vybral. Vyplavalo na povrch určité schéma, které bude nutné dodržet. Pro možnost ovládání simulátoru různými knihovnami strategií bude nejlepší architektura klient-server, kde serverová aplikace bude vlastní simulátor pracující s reálnou fyzikou a vlastní vizualizací. Simulátor se tak stane nezávislým na způsobu či rychlosti výpočtu řídicích dat, které navíc mohou být počítány i na jiném stroji.

Vstupními daty simulátoru budou příkazy pro jednotlivé roboty, které lze asynchronně vyměňovat prostřednictvím definovaného protokolu. Výstupními daty pak budou pozice a orientace robotů. Na základě těchto dat si může řídící knihovna strategií rozhodnout jaké příkazy robotům zaslat příště. Pro podporu vzniku klientských aplikací jsem naprogramoval torso klientské aplikace, které umožnuje vizualizaci stavu simulace a postará se o výměnu dat se simulátorem. Pro uživatele mého simulátoru tedy zbývá, doplnit kód tohoto předpřipraveného torsa o logiku řízení robotů a v momentě, kdy mají příkazy připraveny zavolat metodu, která data zašle zpět do simulátoru. O možnostech řízení se rozepíšu později, předesílám zde jen, že jsem na serverové straně také připravil přídavnou logiku pro podporu řízení robotů.

Zpět k hlavnímu simulátoru. U aplikací, kde se používá OpenGL a nebo her obecně, bývá běžné, že jsou všechny objekty scény spravovány určitým managerem, kde jsou udržovány jejich modely a pozice. Dobrým zdrojem informací může být [4] a [5]. Taková scéna pak musí existovat i pro potřeby simulace v PhysX. Využívat pro vizualizaci přímo scénu určenou pro simulaci v PhysX je sice možné, nicméně nepraktické s ohledem na modularitu systému, protože všechny třídy zabývající se vizualizací objektů by musely mít importovány také knihovny PhysX. Takto implementovaná vizualizace by tak mohla být použita pouze ve vlastím simulátoru. Z dřívějšího však víme, že je zde požadavek na vizualizaci výstupních dat simulátoru také na straně klienta a ten nemusí být vybaven hardwarem s podporou PhysX. Není vůbec důvod, proč by klientská aplikace měla o přítomnosti PhysXu vědět.

Pro každý objekt, který budu ve fotbalu robotů simulovat, bude proto zapotřebí hned dvě vlastní třídy. Jedna bude řešit otázku vizualizace, která bude obsahovat metodu pro vykreslení pomocí OpenGL a druhá bude řešit otázky simulace a bude tak obsahovat metody, které vytvoří svého aktora pro PhysX. Datové prvky, které obě třídy potřebují pro svou funkcionalitu, ale jsou v zásadě společné, jako jsou poloha, orientace objektů, rozměry a podobně, pak budou uloženy a vyměňovány pomocí třídy Datacore, která bude také zajišťovat výměnu dat mezi serverem a klienty.

Aplikace pak při svém startu inicializuje framework PhysX a knihovny OpenGL a pomocí tříd zastupujících simulované objekty se vytovří na základě dat v Datacore jak simulovaná scéna pro PhysX tak i scéna pro vizualizaci. Jednotlivé instance vizualizovaných objektů si s sebou nesou svuj vlastní model. Nevýhodou tohoto přístupu je sice vyšší paměťová náročnost, ale výhodou je snadná správa paměti a možnost

různých modelů pro každého aktora simulace. Všechny modely využívané vizualizací jsou akcelerovány uložením geometrií do listů. Při změnách parametrů jsou tyto listy zrušeny a opětovně sestaveny tak, aby odpovídaly nově nastaveným parametrům. Také entity zastupující simulační část jsou kódovány tak, aby mohly být parametry simulovaných objektů měněny za běhu. K detailům se dostaneme později.

Vlastní běh aplikace spočívá na straně serveru z koloběhu simulace, aktualizace dat a vizualizace, který lze jako jediný za účelem úspory procesorového času vypnout. Koloběh simulace spočívá v přečtení řídících dat (příkazy robotům) na jejichž základě se nastaví požadované tahy motorů pro každého robota. Tyto tahy jsou přeneseny do simulace a je spuštěn výpočet fyziky, který vede k novým pozicím objektů scény.

Během simulování fyziky přímo na grafické kartě, jsou aktualizována data (polohy a orientace robotů) v Datacore z minulého simulačního kroku. Na základě dat v Datacore je provedeno vykreslení modelů, tedy vlastní vizualizace. Data z Datacore se při své aktualizaci rovněž odešlou všem přihlášeným klientům. Vykreslení probíhá iterací přes všechny objekty scény a voláním metody pro vykresleni užitím příkazů OpenGL. Vizualizace zabírá nejvíce času a protože neběží v separátním vlákně je blokující pro spuštění dalšího simulačního kroku. Rychlost simulace může významně zatěžovat i přídavná logika pro podporu řízení robotů, o které budu psát v závěru kapitoly. Předešlu, že příkazy, které robotům zasíláme se můžou lišit úrovní abstrakce. Simulátor počítá v současnosti se dvěma možnostmi. Vstupem do simulátoru mohou být požadované tahy motorů pro každého robota, nebo cílový bod, kam má robot dojet. Kdy simulátor pomocí později popsaných algoritmů vyhodnotí tahy motorů a nastaví je jako síly, motorů, které budou formovat budoucí stav simulace.

Na straně torsa pro klientskou aplikaci je pak koloběh složen jen z vizualizace a aktualizace stavu Datacore. Protože četnost simulací může být významně vyšší, než četnost vykreslování, je odběr pozic a natočení robotů prováděn v separatním vlákně a s každými novými daty je prováděna aktualizace dat třídy Datacore. Vykreslována je pak vždy situace v co nejaktuálnějším stavu. Jinými slovy, je časté, že pozice některých robotů jsou z jiného časového rámce, než zbytek scény. Maximální časový rozestup je však dán délkou vykreslení jednoho snímku a je pouhým okem nepostřehnutelný. Pro toto řešení bylo pochopitelně nutno řešit zámky na datech v Datacore. A jde o kvalitativně lepší řešení než vykreslení robotů dle pozic v jednom časovém rámci. Alespoň část poloh robotů je aktuálnější, než kdyby byla pořízena kopie dat na začátku vykreslování snímku.

4.2 Modelování reality v PhysX

V této kapitole se dozvíte jakým způsobem jsem namodeloval jednotlivé prvky fotbalu robotů. Proč jsem zvolil právě tuto reprezentaci a také jaké výhody a nevýhody to přináší.

4.2.1 Globální konvence

Aby bylo možné začít s modelováním aktorů. Bylo nutné si stanovit pár základních věcí. Simulátor bude simulovat objekty o velikostech v řádech centimetrů. Implicitní nastavení knihoven PhysX umožňuje pronikání dvou objektů z důvody stability simulace až do

úrovně 0.05 základní jednotky. Přičemž jednotky simulace nejsou vynuceny. Parametr, který ovlivňuje pronikání objektů do sebe se jmenuje NX_SKIN_WIDTH. Modelovat tudíž náš svět se základní jednotkou délky 1 metr, by mohlo vést k velmi nepřesvědčivým výsledkům simulace. Představte si že jsou roboti do sebe zaklínění v hloubce 4cm a k odrazu stále nedochází! Snižování parametru NX_SKIN_WIDTH je sice možné, ale dle dokumentace PhysX může docházet k snižování věrohodnosti a stability simulace. Vše bude pravděpodobně zapříčiněno nízkou přesností čísel, s kterými se ve PhysX počítá. Kvůli multiplatformnosti a povaze procesorů, které fyziku počítají je použit jen 32-bitový float.

Bylo proto nutné slevit z požadavků používání soustavy SI v zájmu vyšší přesnosti a základní jednotkou délky stanovit centimetry. Hmotnost pak gram a čas můžeme ponechat ve vteřinách. Se všemi odvozenými jednotkami pak musíme zacházet dle této konvence, takže rychlost bude pochopitelně cm/s. Tyto fakta musíme zohlednit při modelování světa, abychom podvědomě nezadávali rozměry v metrech nebo nečetli vektory rychlosti jako nesmyslně vysoké. Aby bylo tomuto textu správně rozumět, PhysX sám o sobě o použitých jednotkách neví nic, jde jen o konvenci, jak vnímat hodnoty čísel a jak je při modelování nastavovat.

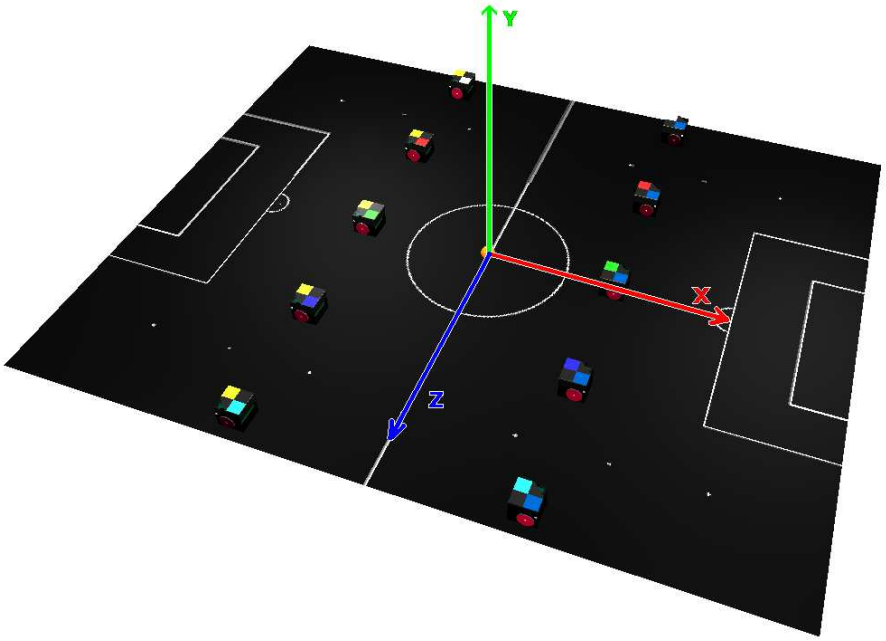
Druhou věcí, kterou si musíme při vytváření scény rozmyslet je soustava souřadnic. Frameworku PhysX je podle dokumentace jedno, zda budeme používat levotočivou, nebo pravotočivou soustavu souřadnic. Vzpomeňme si ovšem na zadrátování výpočtu normál při vytváření meshů. Takže bych si dovolil o tomto tvrzení zapochybovat. Kdo ví, kde ještě bych v budoucnu narazil na problém. Protože k vizualizaci budeme využívat OpenGL, rozhodl jsem se pro tvorbu scén využívat soustavu souřadnic dle OpenGL. Základní náhled tedy bude po ose -Z. Tato konvence nám umožní používat matici globalPose, která určuje polohu tělesa ve scéně jako matici ModelView pro účely vizualizace. Používanou soustavu souřadnic si můžete prohlédnout na obrázku 2.

Na závěr si je potřeba rozmyslet nastavení vytvářených scén. Gravitace bude působit jako síla ve směru osy -Y. Což ze základního pohledu OpenGL bude vypadat jak jsme zvyklí, směrem dolů. Roboti pak budou hrát na rovině XZ. Tomu musí odpovídat i umisťování objektů do scény. Rozměr X bude reprezentovat délku hřiště a rozměr Z šířku. Modelování robotů by pak mělo dodržet stejné konvence. Aby nebylo nutné složitě přepočítávat a rotovat směrové vektory působení sil při přechodu ze souřadné soustavy scény do soustavy robota.

4.2.2 Krokování simulace

PhysX SDK umožňuje provádět simulaci reálné fyziky ve scéně pomocí volání simulate(čas_v_ms). Tato volání se provádí na konkrétní instanci scény a jsou neblokující. Znamená to, že je možné paralelně simulovat více scén. O skončení simulace se uživatel nedozví generováním události. Může se buď opakovaně dotazovat na stav simulace, nebo zavolat metodu fetchResults() na konkrétní instanci scény, která je blokující. Na této instrukci se tedy dané vlákno zastaví, než jsou výsledky k dispozici.

Všechny data scény jsou zdvojeny. Právě metoda fetchResults() slouží k přepsání uživatelského bufferů bufferem pracovním. Tento způsob práce umožňuje, provádět užitečné



Obrázek 2: Soustava souřadnic

nou činnost s daty z minulého kroku simulace i během simulování scény. Pomocí ukazatele na scénu tedy vždy přistupujeme k bufferu, se kterým se v současné době nepracuje. Pokud však chceme stav simulace řídit, nezbývá nám, než nastavit požadované externí síly, které budou formovat stav simulace (jako tahy motorů robotů) dříve, než spustíme metodu simulate(). Po odsimulování jsou totiž vždy externí síly ze scény odstraněny. Čas, kdy máme volný procesor tak lze využít pro účely výpočtů řídicích strategií na starých datech, nicméně jejich aplikace je možná až po přehození bufferů před voláním metody simulate(). V tento moment jsou už ale k dispozici data novější.

Co však lze s volným procesorovým časem podniknout, je vykreslení scény. Vykreslení scény bude s ohledem na složitost simulované scény trvat delší dobu, než simulace. Tento fakt je pro nás žádoucí, protože to znamená, že nebudeme čekat na blokujícím volání fetchResults() a využijeme tak maximální hardwarový potenciál nativně tak, že budeme během výpočty fyziky kreslit. Až se v programu narazí na volání fetchResults(), budou již data připravena na překlopení.

Jedním z nejzajímavějších problémů, které jsem v této práci řešil, je jakým způsobem krokovat simulaci. Jakým způsobem zobrazovat výsledky, aby odpovídaly co nejvíce stavu simulace a aby tak navozovaly dojem běhu v reálném čase a hlavně jak udržet časovou synchronizaci mezi reálným a simulovaným časem. Také mě tížilo aby byla simulace prováděna po dostatečně krátkých časových intervalech a nedocházelo tak k újmám na přesnosti simulace.

Běžná technika určení kroku simulace je odsimulovat čas, za který byl vykreslen předchozí snímek. Je zde však pár problémů. Vykreslení dvou po sobě následujících snímků může trvat jinou dobu, protože se změní obsah vykreslované scény. Zadruhé není tato metoda odolná vůči odejmutí procesorového času aplikaci. Měřil by se totiž jen čas strávený v metodě paintGL() a prodlevy mezi jejich volání by nebyly simulovány. Skutečný a odsimulovaný čas v PhysX by se tak pochopitelně rozcházel. Čas odsimulovaný v aplikaci by byl kratší, než ten, který skutečně uběhl. Čím pomalejší hardware by uživatel simulátoru používal, nebo čím více aplikací by měl spuštěno, tím „pomaleji“ by postupoval čas v simulátoru.

Také použití kroků fixní délky nemá smysl. I když by odpadl problém s udržením časové synchronizace, je problém se stanovením délky onoho fixního kroku. Pokud nastavíme délku kroku příliš malou, nebude moci být výpočet proveden z výkonnostních důvodů. Pokud by byl krok příliš dlouhý, bylo by dosaženo nízké frekvence překreslení a výstup by působil trhaně. Hardware bude tedy buď limitujícím faktorem pro běh, nebo nebude plně využit.

Můj návrh časování pracuje zhruba takto. V hlavní smyčce aplikace probíhá pomocí QTimeru volání metody paintGL(). Četnost tohoto volání závisí na výkonu vašeho počítače, protože Timer je nastaven tak, aby generoval svou událost při idle stavu aplikace. Jinými slovy, pokud je dokončena veškerá práce v daném cyklu, vyžádá si timer překreslení aplikace. V aplikaci se udržuje časová značka posledního odsimulovaného času. Při volání metody pro vykreslení se spočítá rozdíl mezi časem běhu aplikace a časem, který byl odsimulován a tento rozdíl je požadovaný čas simulace, který se předá jako parametr metody simulate() scéně. Touto technikou je zajištěna synchronizace reálného času s během času simulace. Navíc bude dosaženo tak vysoké snímkovací frekvence, jak jen Váš hardware dovolí. Tedy na rychlejším počítači bude rozdíl mezi stavem simulace a vykreslovaným stavem menší, protože simulovaný krok bude kratší a poskytovaný vizuální výstup je vždy 1 krok za simulací. Problematikou real-time vykreslování se zabývá publikace [7].

Zbýval tedy vyřešit poslední problém, a to je kvalita simulace při různě dlouhých krocích. Když jsem v této oblasti důsledně prostudoval dokumentaci k PhysX, zjistil jsem, že interně PhysX provádí simulace po stejně dlouhých časových intervalech bez ohledu na čas, který si žádáte odsimulovat. Rozdíl bývá v tom, kolik těchto interních kroků se provede, než je možné označit stav simulace za konečný. Tento implicitní stav lze změnit pomocí volání metody setTiming() scény, kde lze subkroky nastavit i jinak. Pro účely tohoto simulátoru je však implicitní nastavení vhodné.

4.2.3 Míček

Míček, s kterým se hraje robofotbal MiroSot je standardní oranžový golfový míček. Takový míček váží 46g a má poloměr 2,135cm. Je vyroben z pevného materiálu, takže ho lze modelovat pomocí fyziky pevných těles. Přestože se nebude odrážet tolik jako míček gumový, je tento druh míčku dutý a jeho odrazivost bude poměrně vysoká. Povrch míčku není úplně hladký, takže při svém kutálení bude docházet k vyššímu tření. Také tyto požadavky dokáže PhysX zohlednit a to definováním vlastního materiálu a nadefinování

požadovaných materiálových vlastností. Materiály používané uvnitř frameworku PhysX umožňují nadefinovat dvě základní vlastnosti.

První je restitution, která zohledňuje, jak moc se bude objekt vyrobený tímto materiálem odrážet. Jde o konstantu v rozmezí 0 až 1 a říká, kolik energie se při kolizi zachová jako energie odrazu. Hodnoty blízké 1 simulují opravdu snadno odrazivé objekty jako pingpongový míček. Hodnoty blízké 0 pak jsou vhodné pro hrubou simulaci měkkých objektů, které při nárazu dojde ke konverzi většiny energie na deformace objektu. (Hrubé proto, že framework PhysX umožnuje i simulaci fyziky měkkých těles)

Druhá materiálová vlastnost je pak tření. Různé povrchy těles mohou klást různý odpor při posouvání. PhysX nabízí dvě hodnoty pro zachycení reality v oblasti modelování tření. První je staticFriction, která udává ochotu tělesa, vůbec se do pohybu vydat. Druhou vlastností je dynamicFriction, která definuje k jak velkým ztrátám energie bude docházet během klouzavého pohybu.

Už jste někdy uklouzl na sněhu? Přesně tuto skutečnost lze díky rozdělení tření na statické a dynamické parametry simulovat. Bota má s náledím docela silný kontakt díky sněhu v podrážce, který se přichytí k ledu v momentě našlápnutí. Tuto skutečnost lze zohlednit nastavením vysoké hodnoty staticFriction. Pokud se však síla tohoto přichycení překoná, bota už klouže velmi snadno. Což lze namodelovat pomocí nízké hodnoty dynamicFriction.

Pro namodelování našeho míčku bylo potřeba hodnoty materiálových vlastností experimentálně odhadnout. Statické tření nebylo těžké, kulička nebude klouzat nikdy. Vždy bude mnohem snažší ji uvést do pohybu valivého, proto jsem nechal hodnotu statického tření na 0,5, jako pro simulaci nezajímavý parametr. Dynamické tření jsem nastavil na hodnotu 0,3. Zohledňuji tak fakt, že tření téměř hladkého povrchu golfového míčku bude nižší než průměrné. Nižší hodnoty však jsou vyhrazeny pro simulování snadno klouzavých objektů jako kostky ledu.

Na závěr je potřeba přiznat, že existují objekty s různým třením v různých směrech pohybu. Typickým zástupcem takového objektu bude hrana brusle. Ta dopředu klouže velmi snadno ve srovnání s pohybem do stran. Také dřevo vykazuje významně nižší tření ve směru růstu. Nebo pokud pustíte lyži ze svahu, také dojede špičkou napřed. Aby bylo možné zohlednit v simulaci i takové skutečnosti, nabízí PhysX definici anisotropního materiálu. Kde lze definovat dva páry hodnot tření a to ve směru vektoru směrového a ve směru kolmém ke směru pohybu aktora. Takovýto objekt se během posouvání bude orientovat vždy čelem ke směru pohybu a to tak rychle, jak velký je rozdíl mezi parametry tření. Pokud se parametr tření přežene. Bude se pak těleso spíše valit kolem jedné ze svých os, zatímco v pohybu ve druhém směru bude klouzat.

Výběr tvaru, kteý bude zastupovat míček je zřejmý. Tělo aktora bude složeno z jediného základního tvaru NxSphereShape vytvořeného s poloměrem 2,135cm. Tomuto tvaru bude přiřazen právě definovaný materiál. Pro shrnutí restitution 0,5, staticFriction 0,5 a dynamicFriction 0,3.

Vytváříme aktora dynamického, takže bude nutné vytvořit také body, pro míček. Body musí obsahovat těžiště a hmotnost. Hmotnost míčku nastavíme na 46g. Těžiště můžeme

zadat jako střed míčku, nebo nechat PhysX, at'si těžiště dopočte. Věřím, že výpočet těžiště z jednoho tvaru dopadne dle očekávání ve středu míčku.

Actor pro zastupování míčku je připraven pro nasazení do scény. Pro případy, že budete chtít měnit parametry míčku, byl upraven konstruktor míčku tak, aby vytvářený aktor měl dva volitelné parametry, rozměry míčku a jeho hmotnost. Také byly implementovány metody, které umožní měnit hmotnost a poloměr míčku přímo za běhu aplikace.

4.2.4 Hrací plocha

Hrací plocha[8] může nabývat dvou rozměrů. Podle toho se poměrně zvětšuje i branka. Co je pro nás zajímavé z pohledu fyzikální simulace je povrch hřiště. Povrch hřiště dle pravidel musí zajistit dostatečný grip robotům. Uvedený příklad je povrch pingpongového stolu. Kola robotů jsou v bodech kontaktu pogumovány, takže by naše požadavky měl uspokojit běžný hladký povrch. Protože jde o pevný, neohybný materiál a jeho šířka umožňuje jen minimální pružnost. Bude mít materiál, který pro hřiště použijeme, nízké hodnoty staticFriction, střední hodnoty dynamicFriction a nízké hodnoty odrazivosti. Přesné hodnoty byly určeny experimentálně domácími pokusy odrazy golfového míčku na běžném dřevěném lakovaném stole.

Tvaru hřiště nepůjde dosáhnout jedním základním tvarem. Nicméně jeden tvar NxPlaneShape a několik NxBoxShapů naše potřeby bohatě uspokojí. Protože hřiště bude svou hmotností mnohanásobně překračovat hmotnosti robotů i míčku, není nutné hřiště simulovat jako dynamického aktora, ale postačí, když bude aktor definován jako statický.

Tento aktor by pro simulace bohatě dostačoval. Nicméně, později jsem se ke hrací ploše rozhodl přidat ještě dva NxBoxShape dovnitř branek, s nastavenými flagy, aby se chovaly jako triggery. V budoucnu budeme díky ním snadno vyhodnocovat branky. Velikost těchto triggerů je spočtena tak, aby se dosáhlo kontaktu míčku s objemem zaujímajícím gólovou oblast až tehdy, když míček překročí brankovou čáru svým plným objemem. To je také důvod, proč je pro konstrukci hřiště vyžadován jako vstupní parametr poloměr míčku. Pokud není dodán, použije se implicitně přednastavená hodnota poloměru platná pro klasický golfový míček.

4.2.5 Robot

Na řadu v modelování zástupců pro PhysX se nyní dostal hrací robot. Modelování započalo mnohem dříve, než jsem se dostal ke skutečnému robotu , což vedlo zpočátku k chybám ve věrnosti simulace. Myslím si, že bude zajímavé, když Vám popíšu vývoj avatara hracího robota pěkně postupně, jak vznikal. A seznámím Vás s chybami kterých jsem se dopustil.

Na první pohled se robot jeví jako kostka, která má dvě kola. Tyto kola jsou poháněna nezávislými motory. Nyní parafrázuji dokumentaci k PhysX SDK, kde se píše, že se máme pokoušet o co největší zjednodušení modelů, které předáváme pro výpočet simulace. Ve skutečném světě se konstruuje hodně věcí zbytečně složitě jen proto, že se musí

[8] definována dle FIRA pro fotbal robotů MiroSot

dodržovat veškeré zákony fyziky. Představte si třeba zavěšení kola u běžného automobilu. Kolik různých udělátek musí kolo držet na správné pozici, aby se mohlo otáčet ve všech požadovaných směrech, aby šlo brzdit, pohánět a tlumit otřesy od vozovky. Tyto technická zařízení však není v žádném případě zájem simulovat. Stačí když použijeme jednu vazbu (joint) přímo mezi kolem a tělem vozidla. NxD6Joint nám umožní simulovat omezení v pohybu ve všech požadovaných osách, a kde se kolo nepohybuje, tak pohyb zamkneme. Navíc vlastní osa vazby, mezi kolem a tělesem může vést klidně skrze kolo a vůbec mu v otáčení vadit nebude. Simulace bude s výrazným zjednodušením výpočtů bez jakékoli újmy.

Vybaven touto radou jsem se pustil do modelování robota. První nástřel byl vymodelovat robota jedním tvarem NxBoxShape. Kola přece nejsou nutná, takže aby se robot pohyboval bude stačit aplikování sil na dvou místech, tam kde by byla kola. Směr působení síly pak k čelu robota. Aby se robot pohyboval jako na kolech, tedy ve směru dopředu s podstatně menší ztrátou energie a v ostatních směrech s vysokým třením (simulujícím smyk pneumatik), rozhodl jsem se pro vytvoření anisotropního materiálu. Ve směru pohybu čelem robota pak byly nastaveny hodnoty statického i dynamického tření zanedbatelné, a v ostatních směrech tření s úhlem významně narůstalo.

Po zařazení robota do scény, a krátkém experimentování a pozorování chování tohoto zástupce jsem dospěl k pár závěrům. Robot se pohyboval správně vždy čelem napřed. Síly působící na dvou bodech okrajů robota umožňovaly zatáčení. Setrvačnost byla očekávaná. Odtlačit jiného robota čelně šlo, bočně ne.

Poměrně slibné výsledky zvážíme-li jednoduchost simulovaného zástupce. Nicméně byl zde jeden velice zajímavý jev. Zatímco pohyb vpřed vyžadoval zhruba působení síly o velikosti 1N, pro zatáčení byly potřeba síly zhruba desetinásobné. Tření, které tento zástupce robota produkoval vycházelo jasně vyšší (z důvodu významně větší kontaktní plochy), než při použití kol. Snížení hodnoty tření při pohybu jiným směrem než vpřed sice tento problém odstranilo, nicméně pak zase robot ochotně klouzal i bočně. Trvalo nějakou dobu, než jsem si uvědomil, kde je problém. Pohyb po pneumatice totiž generuje ztráty energie v důsledku valivého odporu, nikoli tření. Tření se aplikuje při změnách směru, nebo rychlosti a parametry pneumatiky v oblasti tření jsou záměrně všesměrově maximální. Způsob, kterým jsem chtěl modelovat chování robota (anisotropním třením u materiálu) nemohl uspět a v tento moment jsem to věděl už i já.

Druhý pokus o vytvoření robota, probíhal jinak. Rozhodl jsem se vymodelovat robota tak, abych mohl mezi kola a robota definovat osy, přes které budou kola držet s tělem, a přes které se bude přenášet na kola točivý moment. K tomuto účelu máme v PhysX vazby (Joints). Vazby však jde vytvářet jen mezi aktory (ne mezi tvary), takže bylo nutné zástupce robota rozdělit do tří aktorů.

Tělo robota bylo opět simulováno tvarem NxBoxShape. Protože aktoři sebou nemohou procházet (pokud nejsou v jiné kolizní skupině) byl tento shape užší o rozměry kol. Tímto jsem se prozatím vyhnul modelování komplexních tvarů těla. Použití jiných kolizních skupin stejně nepřecházelo v úvahu, protože ostatní roboti by procházeli těmito koly taky, to je očividně nežádoucí a pro přiřazení kolizní skupiny každému páru kol a robotu není z důvodu existence pouze 32 kolizních skupin možné. Výřezy, které by byly nutné

pro skrytí kol do těla robota by požadovaly vytvoření nekonvexních meshů. Což jsem věděl, že tyto meshe PhysX nepodporuje na úrovní řešení kolizí, a proto jsem se jim prozatím vyhnul.

Tento postup přinesl mnoho nutných změn taky v subsystémech vizualizace, protože nyní ve scéně PhysX byly objekty (kola robotů), které jsem si vlastně nepřál individuálně vykreslovat. Nakonec pro dodržení základní architektury, mi nezbylo než rozdělit vizualizaci robota rovněž na vizualizaci těla, a vizualizaci kol. Abych i pro kola mohl volat metodu, která je vykreslí stejným způsobem jako pro všechny ostatní účastníky scény.

Pozorování namodelovaného robota mi však nepřineslo příliš důvodu k radosti. Zaprvé se osy (Joints), přes které jsem chtěl přenášet točivé síly, nechovaly dle prvotních očekávání. Robot svou váhou spadl na zem, a osy se vyhnuly i s koly. Pohled jak na rozraženou židli... Tento problém jsem odstranil uzamknutím volnosti pohybu osy ve směru rotace kolem os XZ.

Protože simulace z důvodu stability umožňuje určité nepřesné dodržení vazeb, byla každá osa trošičku jinde. Robot nevypadal souměrně. Pomohlo snížení konstanty, která určuje právě volnost konkrétní vazby. Simulace vazeb byla časově náročnější, ale ne tak, aby to bylo problémem.

Třetím problémem a bohužel fatálním se ukázala schopnost přenosu točivého momentu z kola na povrch hrací plochy. Protože pokud je kolo modelováno válcem, a jde o pevnou geometrii, je styčná plocha pro přenos sil příliš malá. Ani úpravy parametrů tření pro materiál kol nevedla k požadovaným výsledkům. Abych dosáhl přesné simulace, musel bych kolo namodelovat pomocí fyziky měkkých těles. Které však jsou pro simulaci časově o mnoho náročnější. A svou podstatou nepříliš vhodné pro simulaci v reálném čase.

V tento moment mi došly vlastní nápady a začal jsem znovu pátrat v dokumentaci dodané k PhysX SDK. Hledal jsem způsob jak tento problém obejít, nebo ošidit. Při studování příkladů praktického nasazení jsem narazil na tvar NxWheelShape. Který nepatří mezi základní typy PhysXu, nicméně svým významem by mezi ně patřit měl...

NxWheelShape slouží přesně k tomu, co u simulování robotů potřebujeme. U kola lze nastavit průměr, šířku, délku pérování, tuhost pérování (+abstrakce tlumičů), síly motoru/brzd, omezující úhlovou rychlost po ose otáčení, maximální úhel natočení, parametry pneumatik a mnoho dalších užitečných vlastností. Jde o velmi propracovaný tvar, bez nějž by simulování vozidel dobře nebylo možné.

Interně pracuje simulace kola tak, že se vyšle paprsek ze středu kola směrem k povrchu. Až paprsek narazí na jiný objekt scény, provede se výpočet vzdálenosti. Pokud je vzdálenost menší než nastavný poloměr kola, je zde nastolen tvrdý kontakt. Pokud je vzdálenost mezi hodnotou poloměru kola a poloměrem kola + délkou pérování, je vytvořen kontakt přes parametry podvozku (síla kontaktu podle stlačení pérování). A nakonec pokud je vzdálenost větší než poloměr kola a délka pérování, není vytvořen kontakt vůbec.

Tlumiče pérování jsou simulovány pomocí tvrdostí pružin a hodnoty suspension targetValue. Která se může pohybovat v rozmezí 0 až 1. Tato hodnota určuje, jak moc budou stlačeny pružiny v klidovém stavu. U NxWheelShape lze navíc nastavit také

maximální úhlovou rychlost kolem osy otáčení, maximální úhel natočení, točivý moment motoru, hmotnost kola a síly brzd (jako záporný točivý moment). Tyto hodnoty se pak přenášejí podle kontaktu, který panuje mezi kolem a vozovkou.

Zpět k návrhu. Finální reprezentace robota ve scéně frameworku PhysX tedy využívá pro namodelování kol právě tohoto tvaru. Kolo samo o sobě, nekoliduje s tvary vlastního aktora, ani s jinými koly. Koliduje však s jinými aktory scény a přenáší na ně požadované síly. Takže lze pro účely simulace kolo zapustit přímo do těla robota, bez nežádoucích účinků. Není proto nutné, ve PhysX modelovat výkroj ve tvaru těla robota, aby do něj zapadlo kolo. Zapuštěním kol dovnitř geometrie těla robota se odstraní rovněž i další chyba simulace, se kterou jsem se potkal.

Protože kola přenášejí síly na všechny cizí objekty které se dostanou mezi střed kola a vzdálenost o velikosti poloměru kola + parametry podvozku. Kontakt cizího robota s kolem způsoboval v mnoha případech otočení druhého robota na jeho zadní stěnu. Odráželo to skutečnost, že kolo, jak je implementováno připomíná celogumové kolo. Pokud by roboti byli vybaveni celogumovými koly, jistě by k tomuto převracení na zadní stěnu při hře docházelo i při skutečné hře. Proto výrobci robotů zajistili snížení bočního tření kol tím, že je buď kolo celokovové, a pouze hrana, určená ke kontaktu s povrchem je pogumována, nebo je kolo z boku chráněno železným plátováním, které klouže podstatně snadněji než-li guma, a přenos sil je tak značně omezen.

Problém, který zbýval dořešit u simulování robota, je tvar jeho karoserie. Aby mohl robot snadněji tlačit míček, je v něm prohlubeň, do které se míček schová dle pravidel až třiceti procenty svého objemu. Bohužel takto vznikne nekonvexní tvar a ten pomocí PhysX nelze simulovat tak, aby byly automaticky počítány kolize mezi těmito tvary navzájem.

Zde jsem využil techniku rozděl a panuj. Tělo robota jsem rozdělil na 5 konvexních částí. Pro každou konvexní podčást robota jsem pak vytvořil tvar NxConvexShape. A aktor pak sdružuje všech pět těchto tvarů a dvě kola NxWheelShape. A nakonec dva tvary NxSphereShape, které reprezentují stabilizační kuličky na přídi a zádi robota. Touto technikou lze namodelovat šasi robota libovolného tvaru.

Pro vytváření mesh objektů do simulace PhysX využívám Cooking SDK, dodaný spolu s frameworkem PhysX SDK. Na použití frameworku Cooking SDK se podíváme hned v příští části tohoto textu. V kódu je pro případ potřeby vytvoření šasi jiného tvaru připravena metoda, která načte .obj model konvexního tvaru a vytvoří z něj mesh, který PhysX akceptuje. Je na uživateli simulátoru, aby si rozbití svého modelu na konvexní podčásti zajistil a v metodě pro vytvoření aktora opět seskládal voláním metody push_back do pole tvarů aktora. Nepředpokládám však, že by tohoto tahu bylo třeba při simulaci libovolného robota MiroSot.

Parametry konstruktoru robota, pak jsou vlastnosti, které určují roboty. Tedy rozměry těla robota, poloměr kol, šířka kol, zdvih robota, dopředný posuv osy kol a maximální točivý moment, který dokáže jedno kolo robota vyvinout.

Všechny tyto vlastnosti lze měnit i za běhu simulace. Změna rozměrů těla robota způsobí změnu měřítka konstruovaných meshů. Stejně tak je tomu i u kol. Podle poloměru jsou osy kol umístěny v takové vzdálenosti od země, aby byl dodržen zdvih robota. Poloměry stabilizačních kuliček jsou pak shodné se zdvihem robota a jsou z poloviny

zapuštěny do těla robota. Dodržena je i vzdálenost od předních resp. zadních okrajů, aby nepřesáhly vnější rozměry celého robota. Dopředný posuv pak udává, jak moc je posunuta osa kol proti geometrickému středu robota po ose X(dopředu/dozadu). Tato hodnota není žádným způsobem kontrolována, a je možné vytvořit kola i mimo vlastní prostor robota, proto je nutné, aby se při vytváření robotů s jinými, než implicitními parametry používalo zdravého rozumu. Posledním měnitelným parametrem jsou pak síly motorů.

Takto vytvořený robot umožňuje simulaci prostřednictvím PhysX na velmi dobré úrovni realistiky.

4.2.6 Použití Cooking SDK

Framework Cooking SDK je dodáván k PhysX SDK a jeho účelem je vytvářet geometrická data, dále jen meshe, které jsou používány interně při simulacích prostřednictvím PhysX.

Hlavní dva typy mesh objektů, které lze pomocí Cooking vytvořit jsou meshe konvexní NxConvexMesh a pak obecné NxTriangleMesh. Zatímco u objektů typu NxConvexMesh je k dispozici řešení kolizí se všemi ostatními tvary, u NxTriangleMesh jsou kolize vyřešeny pouze proti základním tvarům, NxWheel a NxConvexMesh. Dva objekty NxTriangleMesh by sebou prostupovaly. Toto omezení dávají autoři jasně najevo doporučením užívat NxConvexMesh pro simulování dynamických aktorů scény, zatímco NxTriangleMesh je primárně určen pro účely simulace statických aktorů. Určitý rozdíl je také ve způsobu, jakým se tyto geometrie vytvářejí.

Vytvoření konvexních geometrií

Konvexní geometrie jsou zamýšleny k uspokojení potřeby užití složitějších tvarů, než nabízejí základní tvary prostředí PhysX, ale stále jednodušší a výpočetně efektivnější než obecné geometrie. Hlavně pro fázi detekce kolizí. Maximální počet polygonů pro jednu konvexní geometrii je omezen na 256.

Pro přípravu geometrie je potřeba mít uloženy vrcholy geometrie v paměti, nebo v souboru (jako binární data, jednoduchého pole floatů).

Nyní musíme vytvořit objekt typu NxConvexMeshDesc, kterému předáme údaje o bodech. Datové prvky tohoto objektu jsou počet vrcholů, velikost offsetu, reprezentující souřadnice jednoho vrcholu a ukazatel na pole (nebo otevřený soubor), kde jsou vrcholy uloženy.

Nyní inicializujeme SDK Cooking voláním NxInitCooking(). Mezivýsledky „vaření“ jsou ukládány do mezipaměti, buď v paměti, nebo na disku. Osobně preferuji odkládací prostor v paměti, jehož předností je rychlost. Vytvoříme tedy instanci MemoryWriteBuffer.

Pomocí volání NxCookConvexMesh(*NxConvexMeshDesc,MemoryWriteBuffer) se vytvoří do MemoryWriteBufferu geometrie ve tvaru, který PhysX akceptuje. Nyní tyto data nahrajeme hluboko do paměti frameworku PhysX. K tomu slouží volání createConvexMesh(MemoryReadBuffer(*MemoryWriteBuffer)), která se volá přímo na instanci PhysX SDK a vrací ukazatel na NxConvexMesh, jenž obsahuje geometrii připravenou k použití. Nyní můžeme zavolat NxCloseCooking().

Takto vzniklý objekt je nutné předat jako uživatelská data, pro objekt třídy NxConvexShapeDesc. U tohoto objektu lze provést změnu platnou pro všechny body, jako jsou měřítka, nebo posunutí soustavy souřadnic. Ale především, tento ukazatel půjde přidat k aktorovi stejně jako základní tvar, protože již jde o třídu odvozenou od třídy Shape.

V případě že již žádný aktor vytvořený NxConvexMesh nevyužívá, je potřeba jej manuálně odstranit. K tomu uživateli pomůžou dvě metody. Metoda getReferenceCount() volaná na vytvořenou geometri (NxConvexMesh) pro zjištění aktuálního počtu objektů, které tuto geometrii používají. A pak metoda releaseConvexMesh() volaná přímo na instanci PhysX SDK, kde parametrem je geometrie, jenž chceme odstranit. Pokud nevytváříte geometrie 1:1 s objekty a nerušíte je odpovídajícím způsobem, bude i zde nutno implementovat manažér, který bude vytváření geometrií a jejich korektní odstraňování řešit.

Vytvoření obecných geometrií

Druhým typem geometrií, které lze pomocí NxCooking SDK připravit, jsou pak obecné geometrie, složené z trojúhelníků. Při vytváření je potřeba si pohlídat dvě věci. Normály trojúhelníky se nepředávají explicitně, ale předpokládá se že vzniknou jako crossproduct vrcholů (v1-v0) x (v2-v0). Takže při vytváření modelu je potřeba brát tuto skutečnost v potaz. Mnoho modelovacích nástrojů umožňuje při exportu geometrie přeházet pořadí bodů tak, aby vznikly takto dopočtené normály všechny dovnitř/ven. Detekce kolizí pracuje pouze při nárazu ze strany normály. Druhou věcí, kterou je potřeba dořešit je, aby každý bod byl mezi vrcholy geometrie jen jednou. Vede to totiž k problémům detekce kolizí s vlastními body geometrie. Což má za následek nejen pokles výkonu, ale i nespolehlivé výsledky simulace. Opět existují nástroje, které umožní provést kontrolu na duplicitní, nebo blízké body. Já můžu ze svých zkušeností vřele doporučit Blender, který umí hladce předejít oběma těmto problémům.

Pro přípravu vlastní obecné geometrie je zapotřebí mít v paměti, nebo v souboru (opět jako binární data pole floatů) uloženy vrcholy a také pole indexů, ve tvaru integerů, které budou popisovat z kterých bodů se skládají jednotlivé trojúhelníky. Pro indexy lze použít jak 32bit integery, tak i 16bitové. Knihovny Cooking pak o této skutečnosti informujeme nastavením flagu NX_MF_16_BIT_INDICES.

Když máme v paměti připravená data pro geometrie, můžeme vytvořit instanci třídy NxTriangleMeshDesc a nastavit její datové prvky. Tato třída bude vyžadovat ukazatel na pole vrcholů, počet vrcholu, offset, mezi jednotlivými vektory popisujícími jeden vrchol (obvykle 3*sizeof(float)), ukazatel na pole indexů, počet trojúhelníků a offset v poli indexů, mezi dvěma trojúhelníky. Typicky tedy velikost tří integerů.

Nyní můžeme inicializovat Cooking SDK voláním NxInitCooking(). Vytvoříme si buffer, podle místa, kde chceme geometrii připravit. Pro práci s operační pamětí využijeme MemoryWriteBuffer. Nyní je čas zavolat NxCookTriangleMesh (*NxTriangleMeshDesc, MemoryWriteBuffer). Výsledkem volání bude geometrie, vhodná pro simulaci v PhysX, připravená v bufferu. Abychom tyto data nahráli na své místo v PhysX, užijeme volání createTriangleMesh (MemoryReadBuffer (*MemoryWriteBuffer)). Toto volání se provádí přímo na instanci PhysX SDK. Návratovou hodnotou je pak ukazatel na objekt třídy

NxTriangleMesh. V tento moment můžeme buffer zahodit. A ukončit práci Cooking SDK voláním NxCloseCooking().

Abychom mohli námi vytvořenou geometrii použít při utváření nového aktora, je potřeba ji předat do připravené třídy NxTriangleMeshShapeDesc jako datový prvek meshData. Protože takto mohou vznikat i opravdu rozsáhlé geometrie, na rozdíl od NxConvexMesh, kde je počet stěn tělesa omezen na 256, je u tohoto objektu připraveno i několik způsobů stránkování. Nejlepší volbou bývá nechat PhysX rozhodnout o použitém stránkování automaticky a to nastavením datového prvku meshPagingMode na hodnotu NX_MESH_PAGING_AUTO.

Nyní je již tvar připraven pro připojení k aktorovi. Stejně jako pro NxConvexMeshe, platí i pro NxTriangleMeshe, povinnost odstranit nepoužívané objekty z paměti PhysX SDK ručně. Přestože jsem se snažil podat toto SDK opravdu vyčerpávajícím způsobem, v případě hlubšího zájmu můžete zabrousit do dokumentace Cooking SDK, které se distribuuje spolu s PhysX. Tato kapitola zde není zařazena bezdůvodně. Pokud by vznikla potřeba simulátor upravit na jiný typ simulovaných robotů, bude potřeba vyměnit geometrie, které se do PhysX načítají. Pokud se řešiteli tohoto problému podaří rozsekat model nového robota na pět částí, jak se mi to povedlo u robota MiroSot, pak mu bude stačit pouze vymenit obj modely. V opačném případě by měl mít alespoň hrubou představu jak připravená metoda pro načítání jeho obj modelu do PhysX pracuje, aby předešel případným problémům.

4.3 Grafická část simulátoru

Nyní, když jsou připraveni zástupci pro jednotlivé prvky fotbalů robotů a věci kolem fyziky jsou dořešeny, je čas, začít se zabývat vizualizací.

Z předchozího textu vyplynulo, kdy a jakým způsobem se budou jednotlivé objekty, které chceme vykreslovat, vytvářet. Jejich pozice bude aktualizována samotným během simulace. Orientace pro účely vizualizace půjde rovněž získat přímo z dat simulované scény. Co nám tedy zbývá vyřešit je, jakým způsobem se budou objekty kreslit.

Rámcově bude platit pro všechny objekty zhruba tato sada pravidel. Vykreslení proběhne pomocí příkazů OpenGL, které se budou nacházet v metodě Draw() každé jedné entity simulátoru. Z výkonnostních důvodů, budou všechny objekty mít geometrie předkompilovány v listech. Tyto listy se budou vytvářet při konstrukci objektu. Počet těchto listů bude závislý na počtu materiálů, z kterých se geometrie skládají. V každém listu budou geometrie jen jednoho materiálu. Pro graficky působivější výsledky se místo standardního pipeline OpenGL použije pro vykreslování těchto geometrií shaderů. Protože různé objekty můžou využívat různé shadery, budou také shadery připraveny v čase vzniku objektu.

Nyní bych popsal objekty, které jsem vizuálně namodeloval. K modelování jsem použil freewarový program Blender, který můžu vřele doporučit. Dobrá knížka o Blenderu je [10], u které se hlavně nenechte odradit názvem. Opravdu k něčemu je. Také mohu vřele doporučit online knihu o blenderu, která se nachází na Wikipedii. Můžete ji nalézt na http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro. Vlastní modelování je svým

způsobem taky věda. Kniha, která se touto partí zabývá je [9]. Utility pro načítání textur a obj modelů popíšu později.

4.3.1 Míček

Protože se míček skládá pouze z jednoho materiálu, postačí nám jediný list. Geometrie, která v něm bude předkompilována je glutSolidSphere(). Kde poloměr bude převzat z konstruktoru míčku a počet dělení v obou směrech jsem nastavil na 30 dílů. Toto dělení díky použitému shaderu nelze postřehnout ani při vysokém přiblížení.

V metodě draw(), bude z Datacore vytažena globalPose a lightPose. Pozici potřebujeme k nastavení matice modelview a k přepočítání pozic světel, které vstupují do shaderu. Shaderování totiž probíhá na základním modelu, a po zarotování objektu tak, aby odpovídal pozici globalPose, by mohlo nasvícení vypadat divně. Proto je potřeba na pozice světel aplikovat inverzní matici k matici modelview tedy onu lightPose, a teprve tyto pozice předat do shaderu. Získat inverzní matici z PhysX je několikanásobně rychlejší než pokus o vlastní výpočet.

Pro míček se bude využívat shader, počítající s pozicemi čtyř světel a materiálu. Pokud některé světla nepředáme, budou se jako přírůstky za daný světelný bod přičítat nuly.

Materiál pro míček byl zvolen odhadem, měl by připomínat oranžový lesklý plast. Nastavení materiálu v případě potřeby lze provést modifikací hodnot materialAmbient, materialDiffuse, materialSpecular a materialShiness předávaných do shaderu.

4.3.2 Hrací plocha

Hrací plocha se bude skládat ze dvou geometrií. Prvním geometrií bude bíly plast, který je použit na mantinely hřiště. Tyto mantinely se skládají z několika málo trojúhelníků. Jemnější dělení nemá vzhledem k bílé barvě žádoucí efekt. Odlesky na bílé barvě lze na počítači simulovat jen velmi obtížně, takže jemnější triangulace by vedla pouze ke zbytečné ztrátě výkonu.

Druhou geometrií pak bude vlastní hrací plocha. Ta se předpokládá, že bude černé barvy, matný povrch. Zde budou hry světel a stínů více nápadné, takže nebude modelována jedním quadem. Ale bude rozbita na síť trojúhelníků. Metoda, která vytváří požadovaný list geometrií je součástí třídy a jmenuje se MakePlochaList(). Experimentálně bylo stanoveno dělení plochy na 100dílu vertikálně i horizontálně. Tento parametr lze v metodě MakePlochaList() v případě nutnosti opravit. Nicméně 10 000 trojúhelníku na jednu plochu si myslím, že je poměrně hodně a k vizuálnímu výsledku by již navýšení příliš nepomohlo, naopak výkon by určitě poklesl.

Na tuto geometrii pak bude nanášena textura, která obsahuje značení hřiště. Textura je načítána mým TextureManagerem. Odkaz na instanci manažéru je mezi datovými prvky hřiště. Pro hrací plochu se bude využívat čtyř světelný shader texturovací. Jehož vstupy jsou pozice čtyř světel. Tyto světla není nutné přepočítávat, protože pozice hrací plochy se nemění. Plocha je modelována statickým aktorem. A dále je parametrem shaderu číslo zastupující nataženou texturu. Kromě těchto očekávaných vstupů jsou vstupní parametry shaderu rozšířeny o materialReflexness a materialShiness. Které popořadě udávají jak

moc se světla budou připočítávat k textuře (obdoba modulate) a jak ostrý bude odlesk (mocnina cosinu při výpočtu osvětlení). Díky tomuto shaderu lze modelovat celou škálu materiálových povrchů i přesto, že jsme použili texturu. Nastavení hodnot použitých ve vizualizaci jsem provedl opět experimentálně, pro docílení povrchu černého matného plastu, nebo dřeva s matným lakem.

Texturu pro povrch jsem připravil sám za použití Microsoft Paintbrush ;-)

4.3.3 Robot MiroSot

Protože se mi nepodařilo na internetu nalézt model, který by alespoň rámcově splňoval požadavky simulátoru. Robot se po stránce vizualizace stal nejtužším oříškem. Aby byl grafický výstup pěkný, nezbylo mi nic jiného, než jej osobně vymodelovat.

Teďkom bych se vrátil k vlastní vizualizaci pomocí OpenGL. Robot se skládá z více materiálů, takže bude zapotřebí více předkompilovaných listů. Metodu pro vytvoření vlastních listů a jejich počtu jsem naimplementoval obecně tak, aby byla schopna načíst libovolný model ve formátu wavefront (.obj), zparsovat jej a vytvořit tolik předkompilovaných listů, kolik je v modelu obsaženo různých materiálů. Protože OpenGL při volání glGenLists(počet_listů) vrátí první číslo, za kterým je volných počet_listů listů, stačí ve třídě robota skladovat toto vrácené číslo a počet_listů, které jsme žádali. Počet_listů pak bude logicky tak vysoký, kolik má model různých materiálů. Všechny face, využívající stejný materiál pak budou obsahem jednotlivých listů. Stejným způsobem se vytvoří také listy pro kolo. Které dodávám opět jako samostatný model. Bude snadnější s ním pak operovat. Zejména změny poloh os náprav u robota by v případě jednoho modelu byly nemyslitelné.

Při volání metody draw() se jako u míčku i zde musí převzít globalPose a nastavit matici modelview. Dále pak přepočíst pozice světel ve vztahu k natočení robota, aby byl nasvícen z požadovaných stran. Tyto parametry se předají do shaderu spolu s materiálem pro daný list. Vykreslení celého modelu proběhne v cyklu for přes všechny listy (1list = 1materiál). Uvnitř těla cyklu for se mění pro shader pouze materiálové uniformy. Pozice světel se již znova zbytečně nepočítá.

Po vykreslení listů obsahujících tělo robota se vykreslí dvakráte také kolo. A to obdobným způsobem jako tělo. Akorát je nutné lehce posunout místa vykreslování tak, aby kola zapadla do těla robota. Případně, aby bylo kolo natočeno. Tomu musí opět odpovídat i pozice světel předané do shaderu. Pro vykreslení modelů robota i kol se použije materiálový shader se čtyřmi světly.

4.3.4 Připravené shadery

Všechny objekty v simulátoru jsou vykreslovány pomocí shaderů. Pro vlastní simulátor jsem navrhl hned několik shaderů, které se liší svými požadavky an výkon grafického hardware. Bohužel jsem se z časových důvodů nedostal k tomu, naprogramovat profiler, který by o připravených shaderech rozhodl na základě platformy, na které program spouštíte, takže se nakonec používají pouze dva.

Oba ve své vertexové části odvedou povinnou rutinu pro maximálně čtyři světelné zdroje[9] a ve své fragmentové části vypočítají barvu podle Phongova osvětlovacího a pseudo stínovacího modelu. Proč pseudo Phongův model, protože normály jsou do fragmentové části dopočítávány interpolací normál ve zpracovávaných vrcholech fragment shaderu. Výsledek je nicméně při dostatečné triangulaci modelů docela pěkný a výkonnostně je shader zvádnutelný.

V čem se ale dva nasazené shadery liší, je způsob získání základní barvy. Jeden připravený shader je pro nasazení s materiály. Kdy je základní barvou, která se osvětluje určena vlastnostmi materiálu. Druhý shader je připraven pro nasazení s texturou. Funkcionalita texturovacího shaderu je rozšířena o určité materiálové vlastnosti. Jako je základní intenzita odrazu (reflexness) a velikost odlesků (shiness), které jsou běžněji viděny spíše u materiálů, nežli u textur. Důvodem tohoto kroku je možnost jemného ladění vizuálního dojmu aplikace. Pokud bych tyto parametry do shaderu nepřidal, nepůsobila by hrací plocha požadovaným dojmem. Takto je textura nasvětlována podle požadavků uživatele. Tmavším texturám je možné snížit odrazivost, jak tomu je i ve skutečném světě. U světlejších naopak odrazivost můžeme přidat. Velikost odlesku pak dolaďuje dojem správného materiálu, kde na matných površích bývají odlesky větší, ale slabší. U Kovů bývá odlesk menší a ostřejší. Jeden z mých shaderů si můžete prohlédnout ve výpisu zdrojového kódu 4 a 5. Zde předvádím základní variantu pro pouze jeden světelný zdroj, aby kód nezabral příliš místa.

```
uniform vec4 light0_pos;
varying vec3  light0_dir ;
varying vec3 normal;
varying vec3 eye_dir;
varying vec2 tex_coords;

void main()
{
    gl_Position  = ftransform() ;

    //smer normaly vertexu
    normal = normalize(gl_NormalMatrix * gl_Normal);

    //smer dopadajiciho paprsku na vertex
     light0_dir   = normalize(vec3((gl_ModelViewMatrix * light0_pos)-(gl_ModelViewMatrix * gl_Vertex)
          ));

    //smer k oku - neboli opacny smer k vektoru smerujiciho od oku k vertexu
    eye_dir  = normalize(vec3(-(gl_ModelViewMatrix * gl_Vertex)));

    //souradnice vrcholu v texture
    tex_coords = gl_MultiTexCoord0.xy;
}
```

Výpis 4: Kód vertex shaderu pro objekty s texturou

[9] pokud do shaderu pošlete pouze jedno světlo, bude počítáno jen jedno. Tímto způsobem je možné výkonnost aplikačně doladit v budoucnu bez nutnosti zasahovat do shaderu

```
uniform float materialShiness;
uniform float materialReflexness;
uniform sampler2D texture;

uniform vec4 light0_ambient;
uniform vec4 light0_diffuse;
uniform vec4 light0_specular;

varying vec3 normal;
varying vec3 light0_dir;
varying vec3 eye_dir;
varying vec2 tex_coords;

void main()
{
    vec4 color; //pro michani barvy
    vec3 smer_reflekce;
    float cosalfa;

    const vec4 materialAmbient = vec4(0.1,0.1,0.1,1.0);
    vec4 materialDiffuse = texture2D(texture,tex_coords);
    const vec4 materialSpecular = materialReflexness * vec4(1.0,1.0,1.0,1.0);

    //bez ohledu na zdroj svetla, ambientni barva zde bude vzdy
    color = (materialAmbient * (light0_ambient));

    //smer dopadajiciho paprsku * normala ~ cosinus fi ~ lambert test
    //abychom urcili z normal, zda ma byt fragment osvicen
    //zde je nutno dodat, ze prekazka mezi fragmentem a svetlem se nebere v potaz (neni stin)
    //pokud se nepletu to jiz na urovni fragmentu ani resit nelze :-/
    float cosfi = dot(normal, light0_dir);

    //osvetlene plochy... budou mit navic slozky difuzni a specularni
    if (cosfi > 0.0)
    {
        //smer odrazeneho paprsku * smer kamery ~ cosinus alfa
        smer_reflekce = (2.0 * cosfi * normal) - light0_dir;

        //cosalfa = cos uhlu mezi smerem pohledu a smer odrazu)
        cosalfa = max(dot(eye_dir,normalize(smer_reflekce)),0.0);

        //vypocet osvetleni ambientni+difuzni+specularni
        color += vec4(materialDiffuse * light0_diffuse * cosfi)
                + vec4(materialSpecular * light0_specular * pow(cosalfa,materialShiness));
    }

    //spoctenou barva je barvou fragmentu
    gl_FragColor = color;
}
```

Výpis 5: Kód fragment shaderu pro objekty s texturou

4.3.5 Loader Textur

Pro načítání textur využívám v aplikaci TextureManager, založený na Singleton Texture Manageru napsaném Benem Englishem. Původní zdrojový kód však obsahoval nedostatky, které jsem během doby co jej využívám odstranil. Původní zdrojové kódy se mi již nepodařilo znovu nalézt. Ke své práci vyžaduje knihoven FreeImage. Tyto jsou volně stažitelné v rámci GNU GPL licence.

Hlavní předností tohoto manažeru je snadnost použití. K načtení obrázku, jehož datový formát je v kompetenci knihoven FreeImage stačí zavolat pouze jedinou metodu a to LoadTexture(cesta_k_souboru,číslo_v_evidenci). Zjištění datového typu, rozměrů, formátu barev a natažení textury do OpenGL a nově i vytvoření mipmap zajistí manager. Návratovou hodnotou volání je pak identifikátor objektu textury přímo v OpenGL. Pokud chcete texturu předat třeba do shaderu, použijete vracenou hodnotu. Pro bindnutí libovolné textury, která byla natažena pomocí tohoto TextureManageru pak slouží volání BindTexture(číslo_v_evidenci). Pohodlný úklid všech textur pak zajistí volání UnloadAllTextures().

4.3.6 Loader .obj Modelů

Zdrojový kód pro načítání scén uložených ve formátu wavefront (.obj) vznikl seskládáním a úpravami více tříd neznámých autorů. Podstatné je, že poslední použitá verze v této aplikaci je plně funkční. Umožňuje načíst a rozparsovat obj modely skládající se z trojúhelníků nebo quadů. Pokud budete vyžadovat i face s vyšším počtem vrcholů, je potřeba upravit konstantu MAX_VERTEX_COUNT v souboru obj_parser.h. Toto omezení je zavedeno z důvodů, jakým nakládám s načtenými modely v simulátoru robotů já. Pro Loader není problém načítat face o mnoha vrcholech. Loader umí korektně rozpoznat a přečíst také asociované materiály a textury, pokud jsou .mtl ve stejném adresáři jako .obj. Nebo pokud je u nich relativní cesta. Vlastní implementace se skládá ze 4 hlavičkových souborů a 4 souborů s kódem.

begin

list.h / list.cpp - je implementací seznamu, do kterého lze přidávat prvky, které se vyskytují v obj souborech.

string_extra.h / string_extra.cpp - který poskytuje služby pro porovnání řetězců napřímo a na obsah podřetězce

obj_parser_h / obj_parser.cpp - který obsahuje vlastní logiku rozpoznání obsahu scény a vytovření odpovídajících seznamů.

objLoader.h / objLoader.cpp - Který slouží jako obalující třída obj_parseru. Jeho služby využíváte právě přes metody instance objLoaderu.

Použití loaderu je pak snadné. Po vytvoření instance objektu objLoader stačí zavolat metodu load(cesta_k_souboru). Po dokončení načítání pak instance objektu obsahuje datové prvky podle obsahu načteného souboru. A to v seznamech implementovaných jako

dynamická pole a integerech obsahujících počet prvků v jednotlivých seznamech. Díky tomu je pak snadné procházení v cyklech.

Načtená data se nachází v seznamech vertexList, normalList a textureList a jsou to pole vektorů složených ze tří floatů. Tedy vlastní souřadnice. U textur se poslední složka vektoru nechává volná. V poli faceList pak naleznete záznamy typu obj_face. Ten se skládá z integeru, který informuje o počtu vrcholů tvořících face. Dále material indexu, který udává index v poli materiálů pro nalezení materiálu daného face. A tři pole vertex_index, normal_index a texture_index, které obsahují indexy do polí vertexList, normalList a textureList. Materiálový index pak ukazuje do pole materiálů. Pokud je materiálem textura, je zde uvedena cesta k ní. V opačném případě jsou vyplněny materiálové vlastnosti.

4.4 Možnosti řízení robotů

V simulaci jsou roboti řízeni velikostí točivých momentů, které vyvíjí motor na jednotlivé nápravy. Simulátor umožňuje řídit tahy těchto motorů uživatelem přímo. A to voláním metody control() přímo na instanci robota. Kde vstupními parametry jsou dvě hodnoty float, udávající poměrnou sílů, vztaženou k maximální hodnotě tahu motoru. Tedy hodnoty z rozmezí -1 až 1. Výstupy simulátoru lze získat přímo ze simulované scény. Tyto volání je potřeba provést vždy před každým simulačním krokem.

Během mé práce na simulátoru jsem se dostal ke knihovnám, které měly poskytovat umělou inteligenci robotům a to ve formě generování bodů, kam se mají roboti na základě momentálního umístění přemístit. Pro výměnu dat scény mezi simulátorem a knihovnou zde byla nadefinována datová struktura Environment. Obsahem struktury jsou byly pozice robotů, míčku, rozměry hřiště a odhad příští pozice míčku. Tato struktura byla vytvořena při konstrukci scény, aby mohla být poslána do knihoven. Nicméně od využití těchto knihoven jsem byl nucen ustoupit, protože se ukázaly jako chybové a nedokončené. Simulátor je nicméně připraven služeb těchto knihoven využívat v případě, že by vznikla pracující verze. Maximálně by bylo potřeba předefinovat rozhraní v případě, že by došlo ke změnám datové struktury Environment ze strany knihoven.

Tento neúspěch tvůrců knihovny strategií vedl k refaktorizaci značné části kódu mého simulátoru během posledního měsíce vývoje. Protože jsem z původního návrhu systému který počítal s interním nasazením strategie přímo jako knihovny, a který již v tomto textu nebudu popisovat, přešel na model klient-server. Popis mnou definovaného komunikačního protokolu zařadím až za tuto kapitolu. Předpřipravené torso klientské aplikace tento protokol plně podporuje a uživateli pak stačí pouze volat příslušné metody.

Protože je simulace řízena bezprostředně tahy motorů a nasazení těchto knihoven vyžadovalo řízení pomocí cílového bodu, kam má robot směřovat. Bylo nutné navrhnout algoritmus pro výpočet a sledování odpovídající dráhy na takové úrovni aby byly do simulace přeposílány pouze odpovídající tahy motorů.

První verze tohoto algoritmu spočívala ve výpočtu úhlu, který je mezi cílovým bodem a aktuální orientací robota a nastavení točivých momentů motorů, aby se robot co nejdříve natočil přídí k požadované souřadnici. Pokud byl úhel menší než nastavená mez, byly síly motorů postupně přeorientovávány ze zatáčení na akceleraci. Pokud se robot nacházel ve vzdálenosti nižší, než určitý násobek aktuální rychlosti robota, byl

pak výkon motoru tlumen podle poměru vzdáleností a aktuální rychlosti. Cílem tohoto kroku bylo aby robot zastavil svůj pohyb na zadané souřadnici. Tento druh řízení byl poměrně jednoduchý a neřešil problém kolizí mezi jednotlivými roboty. Proto jsem se pokusil navrhnout algoritmus podstatně robustnější a využít technických prostředků, s kterýma jsem se při modelování fyziky a práci na vizualizaci seznámil. Cílem tohoto druhého algoritmu je aby se robot dostal na požadované souřadnice bez kolize s jiným robotem ve scéně.

Prvním krokem tohoto algoritmu je výpočet nejpřímější dráhy, kterou je schopen robot vzhledem k aktuální orientaci a rychlosti dodržet. K výpočtu této dráhy posloužila Beziérova kubika, u které je známo, že začíná v prvním a končí v koncovém řídicím bodě. Otázkou bylo jakým způsobem vypočíst pozice požadovaných řídicích bodů.

První a poslední bod této křivky byl snadný. Prvním bodem bude aktuální pozice robota. Posledním bodem pak bude cílový bod, který má robot dosáhnout. Aby vypočtená křivka zohledňovala robotovy schopnosti zatáčet, musí být do ní zanesen aktuální pohyb robota. Tedy druhým řídicím bodem této křivky bude aktuální pozice robota, ke které bude připočten n násobek aktuální rychlosti robota ve směru aktuální orientace. N násobkem se rozumí konstanta, která určuje schopnosti robota zatáčet. Do matematického a fyzikálního výpočtu této veličiny jsem se nepouštěl s ohledem na skutečnost, že pro další kroky nemá její pozdější modifikace zásadní vliv. A odhadl jsem ji tedy experimentálně. Aby robot skončil po dosažení požadovaného cílového bodu orientován tak, jak si uživatel přeje, byl jako třetí řídicí bod křivky nastaven cílový bod, od kterého je odečten opět N násobek požadovaného vektoru natočení. N násobek je opět konstanta určující schopnost robota zatáčet.

Nyní, když máme vypočteny řídicí body křivky, je čas dopočíst celou trajektorii. K tomuto úkolu je vhodné využít hardware grafické karty. Protože vykreslování křivek je jednou z akcelerovaných úloh, které lze pomocí OpenGL řešit. K tomuto úkolu jsou navrženy evaulátory. Kde vstupem do evaulátoru je pole řídicích bodů a výstupem pak můžou být třeba body Beziérovy křivky. Protože by však tyto body byly výstupem na monitoru, a my potřebujeme tyto souřadnice dále zpracovávat, je potřeba před vyhodnocováním souřadnic přepnout OpenGL do feedback módu, kdy jsou vypočtené geometrie vraceny zpět uživateli do přednastaveného pole. Používání evaulátorů a feedback módu bylo popsáno v části věnované OpenGL. Po napočítání bodů Beziérovy křivky je potřeba si uvědomit, že tyto souřadnice jsou v souřadnicích na výstupním zařízení. Což je pro nás nežádoucí a proto je potřeba využít funkci gluUnproject(), která tyto souřadnice převede zpět do souřadné soustavy scény.

Počet napočítaných bodů je v programu nastaveno jako konstanta. Toto číslo je záměrně nízké a má sloužit k omezení počtu lomených čar, které trajektorii v tento moment zastupují. Důvodem k tomuto omezení počtu lomených čar je to, že tato křivka bude podrobena zkoumání na průniky s jinými tělesy ve scéně a bude nahrazena novou lomenou čárou, která se těmto průnikům pokud možno vyhne.

K detekci průniku této trajektorie s jinými objekty ve scéně využijeme raycasting. Tento prvek PhysX SDK byl stručně popsán v kapitole věnující se frameworku PhysX. Ve zkratce jen připomenu, že jde o možnost vyslat do scény paprsek definovaný dvěma

krajními body. A podle typu raycastingu je navrácen libovolný, první nebo každý průsečík tohoto paprsku s objekty ve scéně. Z výkonnostních důvodů jsem se rozhodl využít volání, které vrací první průsečík.

Za pomoci raycastingu se projde po částech celá trajektorie. Provede se následující algoritmus pro stanovení nové trajektorie, která již nebude obsahovat kolize. Začíná se ze strany pozice robota. Vždy jsou brány dva sousední body trajektorie. První bod, který je souřadnicí aktuální pozice robota je rovnou umístěn do pole bodů nové trajektorie. Provede se raycasting k dalšímu bodu trajektorie. Pokud mezi testovanými body neleží cizí objekt, je druhý bod rovněž umístěn mezi body opravené trajektorie a posouváme se o jeden bod dále v trajektorii. Provede se raycasting mezi posledním uloženým bodem opravené trajektorie a bezprostředně následujícím bodem dráhy. Pokud není nalezen žádný průsečík, opět je bod přidán k bodům opravené trajektorie. Takto se postupuje až k finálnímu bodů trajektorie.

V případě, že je nalezen průsečík s jiným aktorem scény. Je potřeba přemístit bod ke kterému se právě provádí raycasting. Přemísťování se provádí podle několika základních pravidel. První se musí určit, s jakým typem aktora dojde ke kolizi.

Pokud jde o jiného robota. Je bod iterativně posouván střídavě stále více ve směru a proti směru úhybného vektoru. Úhybný vektor je kolmý k ose Y a ke směru raycastovaného paprsku. Vizuálně jde o uhýbání vlevo/vpravo po dráze robota. Po každé úpravě pozice problémového bodu je znovu proveden raycasting. Jestli změna polohy pomohla, nebo nikoliv. Pokud pomohla, je iterace hledání nového bodu ukončena a nová pozice je uložena do bodů opravené trajektorie.

Pokud se jedná o míček, je bod zásahu posunut o 1mm ve směru paprsku a je vyslán nový paprsek ve stejném směru. Trajektorie vedoucí přes míček totiž není žádný problém, nicméně je potřeba se ujistit, že i za míčkem je volno až k dalšímu řídicímu bodu.

Pokud jde o kontakt s okraji hrací plochy, je řídicí bod posunut o 1/2 XZ úhlopříčky robota směrem do středu hrací plochy. Tímto je zaručeno že nebude generována dráha vedoucí za mantinely hřiště.

Celkově je počet iteraci pro hledání jednoho bodu omezen na 50kroků. Pokud není v tomto časovém rámci nalezen bod, kterým by se mohl robot vydat bez kolidování s jiným objektem scény, je trajektorie ukončena pozicí střetu, od které je odečtena opět 1/2 XZ uhlopříčky robota ve směru raycastingu. Koncový bod trajektorie je takto určen těsně před nepřekonatelnou překážkou v naději že s postupujícím časem se situace na hřišti vyvine tak, že již půjde nalézt trajektorii, která bude bez srážky.

Tímto postupem dosáhneme trajektorii úplnou nebo částečnou směřující k cílovému bodu. Nicméně je zde pouze malý počet dělení, kvůli úspoře počtu provedených raycastů. Z tohoto důvodu je potřeba trajektorii znovu vypočíst jako hladkou křivku. Opět lze k tomuto úkolu využít evaulátory na kartě. Kde vstupem bude místo pole řídicích bodů, pole opravených bodů trajektorie. Vznikne tak finální dráha robota, po které je potřeba robota navést. Zde je již nastaven podstatně vyšší počet dělení. Výstup evaulátoru je opět nutno provést v feedback módu, aby jsme napočítané výsledky měli uloženy k dalšímu zpracování. A opět je nutno převést souřadnice této dráhy zpět do souřadné soustavy scény využitím funkce gluUnproject.

Nyní máme připravenu dráhu jednoho robota. Způsob jakým se bude robot navádět po této trajektorii je obdobný prvnímu algoritmu pro řízení robota k cílovému bodu, ve variantě první, kde se neřešily kolize. Podstatný rozdíl je však v bodu, který je předán tomuto algoritmu jako finální. Bod, který je tomuto algoritmu předán závisí na aktuální rychlosti robota minimálně však je vzdálen 3cm po trajektorii robota. K této vzdálenosti je připočten K násobek aktuální rychlosti, kde K je opět konstanta schopnosti robota zatáčet. Navíc je přibrzďování motorem a zastavení povoleno jen v případě, že je aktuálně zvolený bod, ke kterému je robot naváděn již finálním bodem dráhy.

Celkový algoritmus je navíc ještě malinko složitější. Raycastováním bodů trajektorie by totiž nemohly být zachyceny kolize, kde se roboti střetnou jen částí těla. To z toho důvodu, že raycastování umožňuje pouze nalezení dráhy o šířce raycastovaného paprsku (=0) a skutečný robot je pochopitelně širší. Proto jsem ke všem objektům, kde má smysl kolizi s robotem řešit. Přidal ještě obalující geometrii, která obklopuje daného aktora s bočním odstupem právě 1/2 XZ úhlopříčky robota. Tato geometrie má pak nastavenu nulovou hmotnost a pochopitelně je u ní vypnuto řešení kolizí. Prostor, který tak tento tvar zaujímá z hlediska raycastingu reprezentuje pozice, kde by docházelo ke kontaktu mezi roboty z důvodu jejich skutečných rozměrů.

Tento krok však přinesl určité problémy. Vyplývají z techniky raycastingu a z optimalizací, které jsou pro raycastování v PhysX zavedeny. Paprsek, který začíná uvnitř nějakého tvaru, nehlásí kolizi při opuštění tohoto tvaru. Protože jsem prostor který zaujímá tato průchozí geometrie vytyčil dost velký, aby nedocházelo ke kontaktům rohy robotů při zatáčení, lze za určitých okolností dostat druhého robota bez kolize dovnitř inkriminované oblasti. Například tak, že oba dva roboti jedou vedle sebe a jejich cílové body způsobují křížení trajektorií. V momentě, kdy by se měly detekovat hrozící kolize, jsou již roboti navzájem svými středy, kde začíná raycasting, uvnitř obalující geometrie druhého robota, a kolize není zjištěna. Dochází tak ke kontaktu „tělo na tělo“, která ve skutečném fotbale není považována za faul a ve finále může být toto chování posuzováno jako žádoucí, nicméně, já jej považuji za otázku k zamyšlení, jak tento problém dále řešit.

Omezení velikosti těchto obalujících geometrií na 1/2 nejvyššího rozměru robota by sice znemožnilo přítomnost jiného robota v této oblasti díky zákonu neprostupnosti hmot. Nicméně by vedlo ke špatným výpočtům drah a možnosti srážek robotů na místech kde tento rozměr nestačí.

Řešení těchto typů kolizí programově, přineslo problémy s konvergencí celého algoritmu stanovení dráhy. Robot je střídavě posílán ke kolizi a při určité vzdálenosti je kolize odhalena a robot je donucen odcouvat. Celá situace se pak opakuje. Prozatím tento problém nechávám otevřený. Četnost uvíznutí robota však není vysoká a proto jsem se rozhodl funkcionalitu ponechat přístupnou vývojářům strategií. Pokud zjistí, že se robot nepohybuje tam, kam původně zamýšleli, mohou poslat nové souřadnice.

Simulátor tedy nabízí celkem tři možnosti. Uživatelé mohou řídit roboty přímým posíláním tahů motorů, jakoby šlo o skutečné roboty. Přímým bodem, kde ma robot dojet a to bez ohledu na to, zda stojí něco v ceste. A nakonec bodem, kam se má robot dostat po křivce sestavené a prověřené mnou navrženým algoritmem na vyhýbání kolizí.

Datová struktura, kterou si simulátor vyžaduje pro ovládání je níže. Je jasné, že zde bude typ příkazu, který rozhoduje o tom jakým způsobem chce uživatel robota řídit a pak prostor pro souřadnice i dva tahy motorů. Na základě typu pak simulátor rozhodne, která data má číst a jak s nimi dále nakládat. Součástí příkazu je také výběr robota pro kterého příkaz je a to prostřednictvím ID, které je robotu přiřazeno. O ID se klientská aplikace dozví z příchozích dat.

Parametr type určuje způsob řízení robota. Hodnota -1 v kolonce znamená, že daného robota nechce uživatelská aplikace v této sadě příkazů ovládat. Hodnota 0 pak signalizuje, že požadujeme ovládání přímo prostřednictvím tahů motorů. Hodnota je poměrem maximálního tahu motoru robota. Předpokládá se stejná síla motoru v obou směrech rotace a absence brzdícího ústrojí, tak tomu bylo u modelu, který se mi při modelování dostal do rukou. Hodnota 1 signalizuje přímý pokus o dosažení řídicího bodu bez ohledu na možnost kolize. Nejvyšší úroveň abstrakce pak představuje type = 2, který vypočte pro dosažení bodu dráhu bez kolizí a robota po dráze navede.

Souřadnou soustavu simulátoru jsem již popsal výše, nicméně připomenu zde že hrací plocha se nachází na rovině XZ. Středem souřadné soustavy je střed hřiště. Kdy osu spojující branky tvoří X. Při náběhu aplikace je kladná strana X doprava, záporná doleva. Osa Z pak má kladné hodnoty směrem ke kameře. Roboti domácích jsou rozmístění na levé straně, tedy na souřadnicích -X a orientováni ve směru osy +X.

```
typedef struct
{

   int type; // -1,0,1,2
   int id;
   Vector3f position; //odpovida souradne soustave simulatoru
   float torque_left, torque_right; //hodnota <-1 az +1>
}Command;
```

Výpis 6: Datová struktura pro přenos příkazů

Důvodem možnosti přítomnosti hodnoty -1 v kolonce type je, že ovládání se vysílá přes komunikační kanál v celých sadach pro všechny roboty hostů, nebo domácích. A tímto krokem je umožněn multiplayer. Tedy že jeden tým bude ovládat více klientských aplikací. Třeba i uživatel s myší. Set pak má strukturu definovanou následujícím structem.

```
typedef struct
{
   Command commandset[ROBOPLAYERS_PER_SIDE];
}Commandset;
```

Výpis 7: Datová struktura pro sady příkazů

Přestože komunikačním protokolem se budu zabývat později, předešlu na srozuměnou, že sada příkazů je pro každý tým. Tedy že pokud je v týmu 5 robotů, obsahuje sada právě 5 příkazů. Pro posílání řídicích sad domácím a hostům jsou určeny různé porty. Na příchozí data se provádí kontrola ID, zda opravdu řídíme pouze roboty příslušného týmu. Pokud ID nesouhlasí, je daný příkaz zahozen (ne celá sada, jen příkaz robotovi s vadným ID). Počet přihlášených kontrolerů každého týmu je na simulátoru vidět, takže

si lze podvody ohlídat jednoduše tak, že víme, kolik aplikací pro ovládání svého týmu skutečně využíváme.

Pokud si kladete otázku, proč vůbec umožnovat více řídicích aplikací, tak je odpověď následující. Strategie založené na neuronových sítích je potřeba nějakým způsobem učit. Optimálních výsledků lze dosáhnout tak, že bude každého robota ovládat např. jeden člověk a ten 5 robotů najednou neuřídí. Také velmi sofistikované strategie nemusí být dost rychlé, aby vypočítavali rozumně rychle řídicí povely pro 5 robotů, ale zvládnou třeba jednoho, nebo dva. Simulátor tomu nijak nebrání. Sám jsem se do problematiky řešení pohybu k bodu dostal a vím, že bez akcelerace pomocí raycastu by bylo časově nezvládnutelné počítat třeba 10 robotů pro každý krok simulace. Navíc je jasné, že v komplexní strategii je potřeba řešit více, než jen pohyb k bodu. Například přihrávky, odhady pozic robotů v čase t+1 a podobně.

4.5 Komunikace mezi simulátorem a klientem

Pro výměnu dat je simulátor opatřen pěti TCP servery, které poslouchají na portech base_port plus hodnota daného serveru (0 až 4). Implicitní nastavení base_port je 6000. Tedy na portu 6000 je připraven server pro registraci klienta jako pozorovatele v kódu actors_server. Po připojení na tento port je klientovi pravidelně zasílán (bez vyžádání) stav všech aktorů scény, navržená datova struktůra je níže. Vlastní vlastních výměna dat probíhá prostřednictvím UDP datagramů, aby byla zkrácena prodleva mezi doručením v důsledku réžie TCP přenosu. TCP spojení tedy v tomto případě slouží pouze jako nástroj pro výměnu informací o adrese a portu klienta. Případné chyby přensou se projeví maximálně zkreslením jednotlivého snímku, nebo pozic robota. Nová, korektní souřadnice pak bude doručena velmi brzy a proto tyto chyby nebudou mít zásadní vliv na chod simulace. Pro případ, že tyto chyby budou budoucím vývojářům dělat starost, je v obou třídách Datacore, jak na straně simulátoru, tak na straně klientů v metoďě sendActors/readActors zakomentován kód pro komunikaci přímo přes navázané TCP spojení a UDP se využívat nemusí.

```
typedef struct
{
    Robot robots_away[ROBOPLAYERS_PER_SIDE];
    Robot robots_home[ROBOPLAYERS_PER_SIDE];
    Ball  ball ;
    int  settings_num;
    int  score_home;
    int  score_away;
}Actors;
```

Výpis 8: Datová struktura pro přenos stavu scény

Obsahem strukury Actors jsou dvě sady robotů - domácí a hosté, každý robot je popsán strukturou Robot, která je níže. Dále pak je součástí actors Míček. Struktura míčku bude rovněž vyobrazena níže. Hodnoty aktuálního skóre nemusím popisovat. Settings num slouží k informování klienta, že došlo ke změne konfigurace simulace a je potřeba si

vyžádat novou sadu nastavení. Při běhu se pravidelně posílá aktuální číslo konfigurace. Každá změna jej inkrementuje.

Obsahem datové struktury robot je ID robota, které slouží k jednoznačnému rozlišení robota. Parametry barev slouží jednak pro potřeby tvorby modelů vizualizace. Barvy jsou pouze indexy, takže lze u každého klienta barvu nastavit individuálně. Z časových důvodů však toroso klientské aplikace nemá pro tyto účely navrženo GUI. Pozice a natočení robota slouží pro potřeby výpočtů strategií. Matice globalPose a lightPose odpovídají maticím modelview pro vizualizaci, aby se nemusela dopočítávat z pozice a rotace. LightPose pak představuje transformační matici pro polohy světel, které využívají shadery. Ve své podstatě jde o inverzní matici k globalPose, ale je stanovena pomocí frameworku PhysX rychleji než výpočtem na straně klienta.

```
typedef struct
{
    int id;
    int team_color;
    int color;
    Vector3f position;
    Vector3f rotation;
    Matrix4f globalPose;
    Matrix4f lightPose;
}Robot;
```

Výpis 9: Datová struktura pro přenos stavu robotů

Ve struktuře míčku si můžete povšimnout absenci natočení. Která pro výpočet strategií zřejmě nemá smysl. Pozicování do scény a inverzní matice je pak rovněž stanovena již simulátorem a proto může být pro účely vykreslení k užitku.

```
typedef struct
{
    Vector3f position;
    Matrix4f globalPose;
    Matrix4f lightPose;
}Ball;
```

Výpis 10: Datová struktura pro přenos stavu míčku

Další TCP server, který simulátor má, je pro posílání nastavení. Číslo portu Settings serveru je implicitně base_port +1 = 6001. Na tento port se klientská aplikace připojí. Server jí zašle zpět strukturu obsahující nastavení simulace a ukončí spojení. Nepředpokládá se, že by se změny nastavení aplikací prováděli obzvláště často a proto není třeba udržovat zbytečně stavy spojení.

```
typedef struct
{
    int settings_num; //kazda zmena konfigurace navyssi toto cislo
    float domaci_max_toc_moment_motoru;
    float domaci_polomer_kola;
    float domaci_sirka_kola;
    float domaci_dopredny_posuv_osy;
    float domaci_zdvih_robota;
```

```
    float domaci_rozmer_x;
    float domaci_rozmer_y;
    float domaci_rozmer_z;
    float hoste_max_toc_moment_motoru;
    float hoste_polomer_kola;
    float hoste_sirka_kola;
    float hoste_dopredny_posuv_osy;
    float hoste_zdvih_robota;
    float hoste_rozmer_x;
    float hoste_rozmer_y;
    float hoste_rozmer_z;
    float hriste_rozmer_x;
    float hriste_rozmer_y;
    float hriste_rozmer_z;
    float hriste_okraje_x;
    float hriste_okraje_y;
    float hriste_okraje_z;
    float hriste_sirka_branky;
    float hriste_hloubka_branky;
    float micek_polomer;
}Settings;
```

Výpis 11: Datová struktura pro přenos nastavení

Komunikace s kontroléry hostů a domácích a konfigurátory simulace je založena na trvalém TCP spojení. Pro kontoléry domácích je připraven server na portu base_port + 2 (implicitně 6002) pro kontroléry domácích base_port + 3 (implicitně 6003) a pro konfigurátory (to jsou ti, kteří mají právo měnit parametry simulace) je připraven port 6004. Po připojení na uvedené porty se předpokládá opakované (časově nevynucené) zasílání datových struktur zobrazených výše. Serializace dat se provádí podle pořadí parametrů uvnitř definovaných struktur a využívá se QDataStream ve verzi 6_6_0. Obsahem posílaných dat nejsou žádné korekční značky.

5 Závěr

Cílem této práce bylo navrhnout a implementovat simulátor robotů, který bude nejen provádět simulaci reálné fyziky, ale také bude poskytovat grafický výstup. Cíle, které jsem si stanovil jsem také splnil. Mimo rámec zadání jsem navíc simulátor vybavil přídavnou umělou inteligencí pro roboty, která může vývojářům strategií pro roboty fotbalisty významně ulehčit práci. Také jsem pro tyto vývojáře připravil torso klientské aplikace, takže se nemusejí zabývat výměnou dat se simulátorem na nízké programovací úrovni, ale bude jim dostačovat když rozšíří o své strategie připravenou aplikaci a využijí jejich volání na slušné úrovni abstrakce.

V průběhu práce jsem se seznámil s frameworkem PhysX, pokročilými funkcemi OpenGL, jako jsou evaulátory či feedback mód. Zvládl jsem programovací jazyk GLSL pro tvorbu shaderů. Značně jsem se zdokonalil v používání programovacího jazyka C++. Naučil jsem se vytvářet modely prostřednictvím volně dostpuného 3D modeleru Blender. Osvojil si práci s knihovnami Qt. Napsal jsem program, pro načítání obj modelů. Když bude chtít uživatel simulovat jiné typy robotů než MiroSot stačí mu tak k výměně modelů pouze nahrát nový obj model na místo stávajícího. Vlastnosti simulace jako jsou rozměry robotů, síly motorů, nebo posuvy os jeich náprav lze měnit nejen prostřednictvím konfigurace, ale i realtime za běhu aplikace. Kdy simulátor jako výchozí hodnoty používá hodnoty přečtené ze souboru settings.txt. Tento soubor je textový, takže jeho editace nezabere žádné zváštní úsilí a používání celé aplikace se tak stává snadným. Všechny části softwarového díla jsou napsány zodpovědně a robustně a to v naději, že budou v budoucnu používány a rozšiřovány. Tato práce pro mě byla velmi poučná a byla mi na škole doposud největším přínosem.

6 Reference

[1] Shreiner, The Khronos OpenGL ARB Working Group, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Addison-Wesley Publishing Company, 2009.

[2] Rost, *OpenGL® Shading Language, Second Edition*, Addison Wesley Professional, 2006.

[3] Wright, Lipchak, Haemel, *OpenGL(R) SuperBible: Comprehensive Tutorial and Reference*, Addison Wesley Pearson Education, 2007.

[4] Gregory, Lander, Whiting, *Game Engine Architecture*, A K Peters,Ltd., 2009.

[5] McShaffry *Game Coding Complete, Third Edition*, Course Technology, 2009.

[6] Ericson, *Real-Time Collision Detection*, Elsevier Inc., 2005.

[7] Akenine-Moller,Haines, Hoffman *Real-Time Rendering, Third Edition*, A K Peters, Ltd., 2008.

[8] Bailey, Cunningham *Graphics Shaders: Theory and Practice*, A K Peters, Ltd., 2009.

[9] Russo, *Polygonal Modeling: Basic and Advanced Techniques*, Wordware Publishing,Inc. , 2006.

[10] Gumster, *Blender For Dummies*, Wiley Publishing,Inc. , 2009.

[11] Blanchette, Summerfield, *C++ GUI Programming with Qt 4 (2nd Edition)*, Trolltech ASA , 2008.